R Introduction Dyn2

Martin Wegmann

30.04.2018

Table of contents

Built with R version 3.4.2

Syntax Basics

In general you do not have to use a `print()` command to see the result of a command or the content of an object printed on your screen. R is very intuitive in this regard. An exception would be within a loop process or if you execute a script from the command line. Usually R will not print all the commands or results on your screen within a loop chain. If you wish to see part of the results, you can then use `print()`.

In R you can create and fill objects either by using the = between object and content or an **<-**, which defines the direction of the relationship between object and content.

So at this point we will create an object filled with a couple of random numbers:

```
vector_of_numbers<-c(1,4,6,21,56,12,9,85,12)
vector_of_numbers # without print()
```

```
## [1]  1  4  6 21 56 12  9 85 12
```

```
print(vector_of_numbers) # with print()
```

```
## [1]  1  4  6 21 56 12  9 85 12
```

As you can see just typing **vector_of_numbers** is enough to see the content of your object.

The following commands will produce the same objects as above. Note the different use of = and **<-**. Also note that you overwrite objects by default.

```
vector_of_numbers=c(1,4,6,21,56,12,9,85,12)
c(1,4,6,21,56,12,9,85,12)->vector_of_numbers
vector_of_numbers
```

```
## [1]  1  4  6 21 56 12  9 85 12
```

In case you forgot what a command does or you need more detailed insights into how a command is structured, use R's internal help page, either by using `?yourcommand` or `help(yourcommand)`.

With this little vector we can already explore a few common and basic commands such as:

```
length(vector_of_numbers) # gives you the length of your vector
```

```
## [1] 9
```

```
vector_of_numbers*3 # simple multiplication, works for other operations the same way
```

```
## [1]   3  12  18  63 168  36  27 255  36
```

```
mean(vector_of_numbers) # mean of your values in your vector
```

```
## [1] 22.88889
```

```
sd(vector_of_numbers) # standard deviation of all values in your vector
```

```
## [1] 28.52387
```

```
sin(vector_of_numbers) # trigonometric function of all values in your vector
```

```
## [1]  0.8414710 -0.7568025 -0.2794155  0.8366556 -0.5215510 -0.5365729
## [7]  0.4121185 -0.1760756 -0.5365729
```

```
vector_of_numbers[3] # shows you the value at position 3
```

```
## [1] 6
```

```
vector_of_numbers>10 # gives you a TRUE (1) or FALSE (0) binary answer
```

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

Often times in climate science, you need create large vectors in quick ways. To quickly create sequences or vectors with repeated values use:

```
rep(10,times=5) # repetition
```

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

```
1:15 # easy way to create a sequence with default delta of 1
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
```

```
seq(from=0,to=20,by=2.5) # more sophisticated sequence
```

```
## [1]  0.0  2.5  5.0  7.5 10.0 12.5 15.0 17.5 20.0
```

```
# or to create a matrix
matrix(0,nrow=4,ncol=3)
```

```
##      [,1] [,2] [,3]
## [1,]    0    0    0
## [2,]    0    0    0
## [3,]    0    0    0
## [4,]    0    0    0
```

Data read in and overview

Most of the time in climate science you will read in data in ASCII or netcdf format. Either way you might want to get a quick overview of the data inside the file you just read into R.

As usual for command line tools, we should make sure that we are in the desired folder on our machine. This will be the folder where we import data from, but also where we export data or plots to:

```
wd="/Users/mwegmann/Documents"
setwd(wd) # first of all, let us set our working directory
getwd() # this way you can check your working directory in case you forgot
```

```
## [1] "/Users/mwegmann/Documents"
```

For ASCII data the `read.table()` command is usually good enough, since it can handle a lot of file structures. For .csv the `read.csv()` command works very well, although `read.table()` can actually handle .csv data as well. It is also possible to read a file straight from a complete URL.

For the sake of this exercise, let us assume we read in the internal R dataset `iris`. In the following, I just want to hightlight a few quick ways to have a look at your read in data:

```
# so in theory we would read in our data similar to this:
# iris=read.table("iris.txt",head=TRUE)
# but in practice we will just load in the internal dataset
data("iris")

# have a quick look at the structure of your data
str(iris)
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

The output shows us that our data is in a data frame, containing 150 observations times 5 variables. Four of them are numeric, whereas one is a factor with three levels. It also shows you the first few values of each variable. To get a more intuitive overview of a few data points, you can use the following commands:

```
# have a quick look at the first few rows of your data
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

```
# have a quick look at the last few rows of your data
tail(iris)
```

```
##     Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
## 145          6.7         3.3          5.7         2.5 virginica
## 146          6.7         3.0          5.2         2.3 virginica
## 147          6.3         2.5          5.0         1.9 virginica
## 148          6.5         3.0          5.2         2.0 virginica
## 149          6.2         3.4          5.4         2.3 virginica
## 150          5.9         3.0          5.1         1.8 virginica
```

```
# let R show you the most important statistics of your data
summary(iris)
```

```
##   Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
##  Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100
##  1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
##  Median :5.800   Median :3.000   Median :4.350   Median :1.300
##  Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
##  3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
##  Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
```

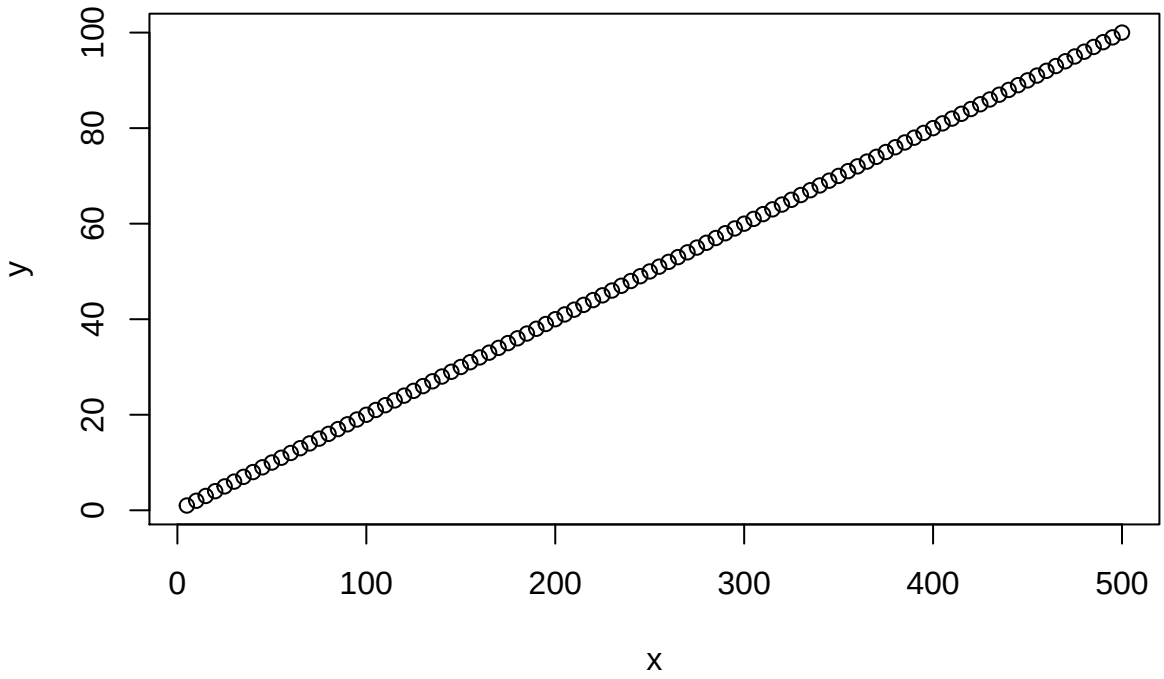
```
##        Species
##  setosa    :50
##  versicolor:50
##  virginica :50
##
##
##
```

Of course there are many more things you can do with your data. For now that should be enough of an overview.

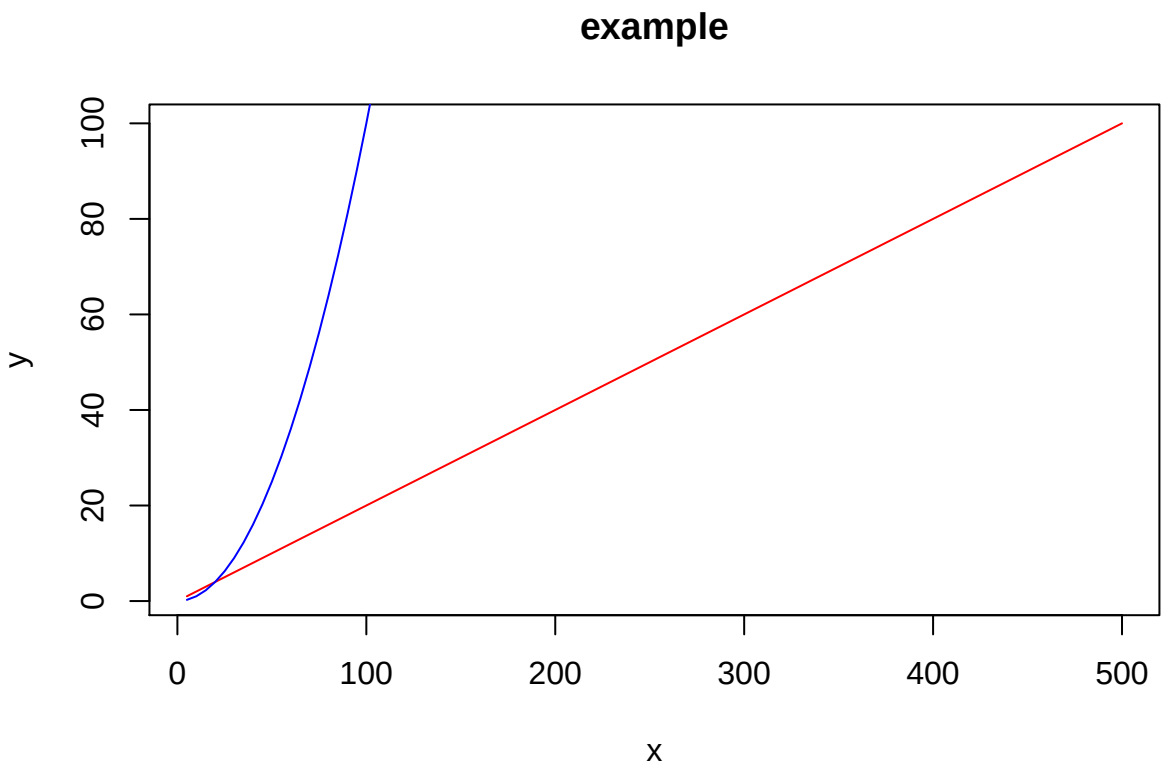
Modifying XY plots

Simple plots are build up as `plot(x,y)`, where the plotting happens in a XY 2D coordinate system:

```
y<-1:100
x<-(1:100)*5
plot(x,y) # a simple plot() is build up as plot(x,y)
```

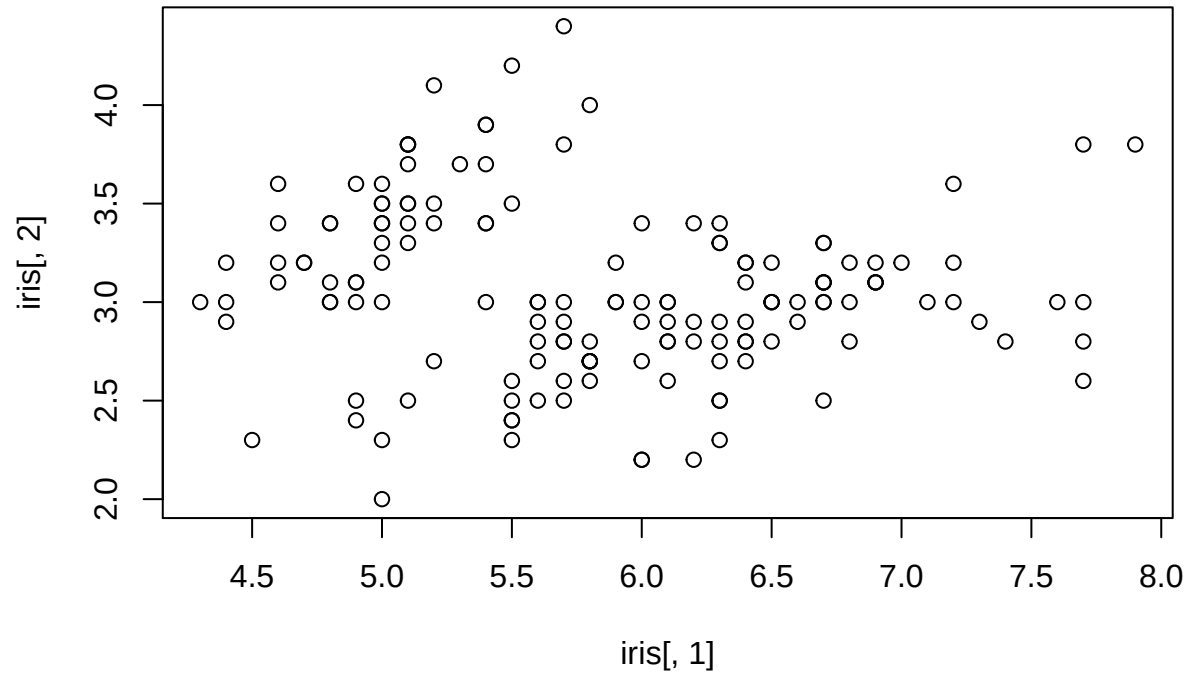



```
# we can add lines, change colors, and describe the plot
z<-x^2 / 100
plot(x,y,col="red",main="example",type="l")
lines(x,z,col="blue")
```



We can now also go ahead and plot variables of our `iris` dataset in a nicer, bigger plot and change the appearance of the plot to suit our needs:

```
plot(iris[,1],iris[,2]) # plot() is build up as plot(x,y)
```



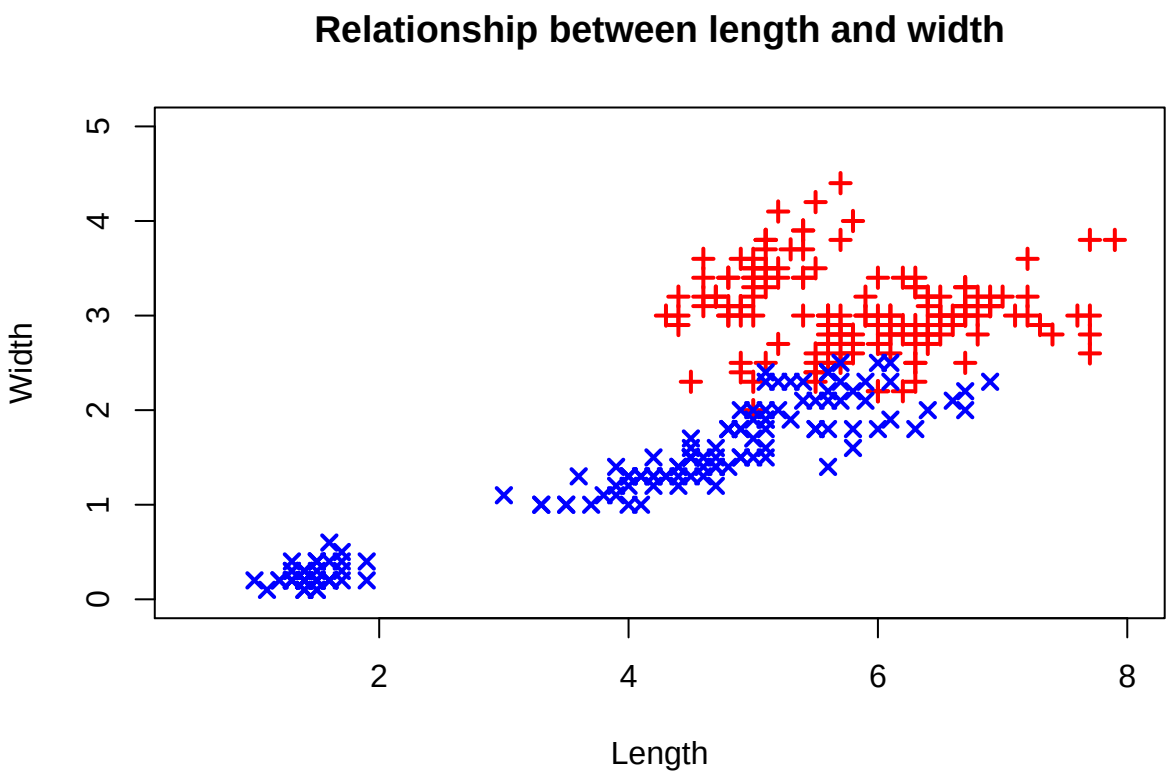
To change or extent a command, R uses small little objects within the `()` frame of a command. The most important for the command `plot()` are:

- `main=` , which allows you to set a headline for your plot

- `ylab=` and `xlab=` , which allow you to label your y or x axis
- `ylim=` and `xlim=` , which allow you to limit your y or x axis to a defined extent
- `col=` , which allows you to set a color for your plot
- `lwd=` , which sets the width of the outline of your plotted object (cirle, line, triangle etc)
- `pch=` , allows you to change the symbol plotted from the default hollow circle to a wide array of symbols.

There are many more parameters and you can dig into all of them using `?par`. But let us now quickly change the plot above to a more colorful and complete version:

```
plot(iris[,1],iris[,2],
     col="red",
     lwd=2,
     ylab="Width",
     xlab="Length",
     main="Relationship between length and width",
     pch=3,
     ylim=c(0,5),
     xlim=c(0.5,8))
# similar to lines() we can add points with points()
points(iris[,3],iris[,4],
      col="blue",
     lwd=2,
     pch=4)
```



Export tables and plots

After we finished our analysis, we often times want to save the results in form of an ASCII file or in form of graphs and plots.

To write our `iris` dataset we can just simply use the `write.table()` command:

```
write.table(iris,"iris.txt")
```

To export plots from R, you have a wide array of choices for the data type. Typical choices include

- `bmp()`, classic windows format
- `jpeg()`, compressed but lossy
- `png()` , lossless
- `tiff()` , lossless and can include meta data, often used in GIS software
- `postscript()` , Adobe language, programmable and vector graphic
- `pdf()` , Adobe language and vector graphic

I personally prefer `pdf()` since it is easily readable and I can scale it up (for example for posters) if I need to. Exporting a plot with `pdf()` would look like this:

```
pdf("histogram_sepal_length.pdf")
# note that by default this plot will be square.
# Adapt width and height in pdf() to change that
hist(iris[,1],main="Histogram of sepal length", xlab="sepal length",freq=FALSE)
dev.off() # to indicate the end of what should be exported
```

```
## pdf
##   2
```

Simple loops

Loops are an essential tool in working with climate data. This chapter here just wants to introduce the general syntax for loops, since different programming languages have different symbols and procedures in loops.

A simple **for** loop in R would look like this:

```
for (i in 1:9) {
  print(i+2)
}
```

```
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
## [1] 11
```

As mentioned before, during loops we need to use `print()` to see the output. You could fill up an empty vector with your new values like this:

```
empty_vector=rep(0,9)
empty_vector
```

```
## [1] 0 0 0 0 0 0 0 0 0
```

```
for (i in 1:9) {
  empty_vector[i]=i+3
}
empty_vector
```

```
## [1]  4  5  6  7  8  9 10 11 12
```

Besides **for** loops, common loops include **while**, **if** and **else**. Simple if/else and while loops would look like this:

```
# if/else loop

if (sum(empty_vector)>10){
print("success")
} else {
  print("failure")
}
```

```
## [1] "success"
```

```
# while loop
i <- 1
while (i < 6) {
print(i)
i = i+1
}
```

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

It is very easy to write a function in R yourself. In this way you can write functions that are perfectly fitting for your analysis and data. An example where I want to see means and standard deviations derived from my input data would be:

```
meansd<-function(x){
    y<-mean(x)
    v<-sd(x)
    list("SD= "=v,"Mean= "=y)
}

meansd(iris[,1])
```

```
## $`SD= `
## [1] 0.8280661
##
## $`Mean= `
## [1] 5.843333
```

Installing libraries

To make the more sophisticated tasks easier, we have to install and then load so-called libraries in R. Libraries are essentially little add-ons for R which extend the functionality or make certain things easier for the user.

The difference between installing and loading is, that we only need to install a package once, whereas we need to load it everytime we restart R.

To install packages/libraries use `install.packages("")`. For example to install the library `ncdf4` the command would be `install.packages("ncdf4")`. Most probably R will ask you for a server mirror to choose from to download the data. Choose a server geographically close to you to ensure a high download speed.

Once this is done, we can load in the library with

```
library(ncdf4)
```

It is advised to put these commands at the beginning of each script to ensure all commands and functions are working.