A gentle introduction to R

GAO Zheng

Dec 25, 2016

Lab info

- Lab section: TBD (GAO Zheng, gaozheng@umich.edu)
- Office hours: (Science Learning Center, Chemistry building)

  TBD

I am roughly following the **TutorialsPoint.com**'s R tutorial: http://www.tutorialspoint.com/r/

This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see http://rmarkdown.rstudio.com.

Getting started

Start by installing R and Rstudio on your computer.

Wroking directory

An R session keeps track of your working directory. All reads/writes/changes/etc. to files living on your computer (hard drive) happens under this directory, unless otherwise specified.

```r
# get working directory
getwd()
```

```
## [1] "/home/gaozheng/Teaching/Stats414/R-intro"
```

```r
# set appropriate working directory (YOUR OWN DIRECTORY)
setwd("/home/gaozheng/Teaching/Stats414/R-intro/")
# Looking around, list all files under this directory
list.files()
```

```
## [1] "data"                       "missfont.log"
## [3] "Stats414_R-intro.knit.md" "Stats414_R-intro.nb.html"
## [5] "Stats414_R-intro.pdf"     "Stats414_R-intro.Rmd"
## [7] "Stats414_R-intro.utf8.md"
```

```r
# alternatively
dir()
```

```
## [1] "data"                       "missfont.log"
## [3] "Stats414_R-intro.knit.md" "Stats414_R-intro.nb.html"
## [5] "Stats414_R-intro.pdf"     "Stats414_R-intro.Rmd"
## [7] "Stats414_R-intro.utf8.md"
```

Look for variables in environment

Within an R session, objects exist in an environment (living in your computer's memory). The one you will be working with most of the time is called the global environment. To

```r
# list variables in the environment
ls()
```

```
## character(0)
```

```r
# create a number in the global environment
a.number <- 12
# look at the environment again
ls()
```

```
## [1] "a.number"
```

```r
# remove a variable from the environment
rm(list='a.number')
# Now 'a.number' is gone
ls()
```

```
## character(0)
```

Data types

There are 6 types of basic data types in R: Logical, Numeric, Integer, Complex, Character, Raw

```r
v <- TRUE
print(class(v))
```

```
## [1] "logical"
```

```r
v <- 23.5
print(class(v))
```

```
## [1] "numeric"
```

```r
v <- 2L
print(class(v))
```

```
## [1] "integer"
```

```r
v <- 2+5i
print(class(v))
```

```
## [1] "complex"
```

```r
v <- "TRUE"
print(class(v))
```

```
## [1] "character"
```

```r
v <- charToRaw("Hello")
print(class(v))
```

```
## [1] "raw"
```

Vectors and Lists

Vectors contains objects of the *same* type. List doesn't require members to be of the same type.

Vectors

```
# initialize a vector (concatenation)
vector1 <- c(1,'A',T,2.3)
# print a vector (works inside functions)
print(vector1)
```

```
## [1] "1"    "A"    "TRUE" "2.3"
```

```
# another way to display a variable (doesn't work inside functions)
vector1
```

```
## [1] "1"    "A"    "TRUE" "2.3"
```

```
# another way to initialize a vector
x <- 1:10
# yet another way to initialize a vector
x <- seq(from = 1, to = 10, by = 1);x
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
# you can leave out the paramenter names as long as the orders are correct
x <- seq(1,10,1);x
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
# initialize a variable using existing ones
y <- x^2
# Notice that you don't have to write a loop to define y
# R is 'vector-friendly'
# In fact you should AVOID writting loops in R unless necessary
print(y)
```

```
##  [1]   1   4   9  16  25  36  49  64  81 100
```

Lists

```
# Create a list.
list1 <- list(c(2,5,3),21.3,sin)
# Now things inside list1 do NOT get converted to the same type!
print(list1)
```

```
## [[1]]
## [1] 2 5 3
##
## [[2]]
## [1] 21.3
##
## [[3]]
## function (x)  .Primitive("sin")
```

Subsetting a vector / list

```
# You can ``access'' any elements of a vector / list by providing its index inside the [ ]
vector1[1]
```

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

```r
# Or by providing a vector of indices
vector1[1:3]
```

```
## [1] "1"    "A"    "TRUE"
```

```r
# Or by providing a vector of Logicals
index <- c(T,F,F,T)
vector1[index]
```

```
## [1] "1"   "2.3"
```

Q: What happens if the length of the Logical index and the vector disagrees?

R recycles the vector with shorter length

```r
vector1[c(T,F,F)]
```

```
## [1] "1"   "2.3"
```

Subsetting a list is a little tricky.

```r
# Subsetting a list with [ ], and you still get a list
list1[1]
```

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

```r
class(list1[1])
```

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

```r
# To extract the vector, use double [[ ]]
list1[[1]]
```

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

```r
class(list1[[1]])
```

```
## [1] "numeric"
```

Pictorial illustration tweeted by Hadley Wickham:

Matrices

Think of them as "2-D vectors". It, too, only allows data of the same type.

Creating a matrix

```r
M <- matrix( c(4,2,6,3,7,1), nrow = 2, ncol = 3, byrow = TRUE)
print(M)
```

```
##      [,1] [,2] [,3]
## [1,]    4    2    6
## [2,]    3    7    1
```

```r
# Matrix transpose
t(M)
```

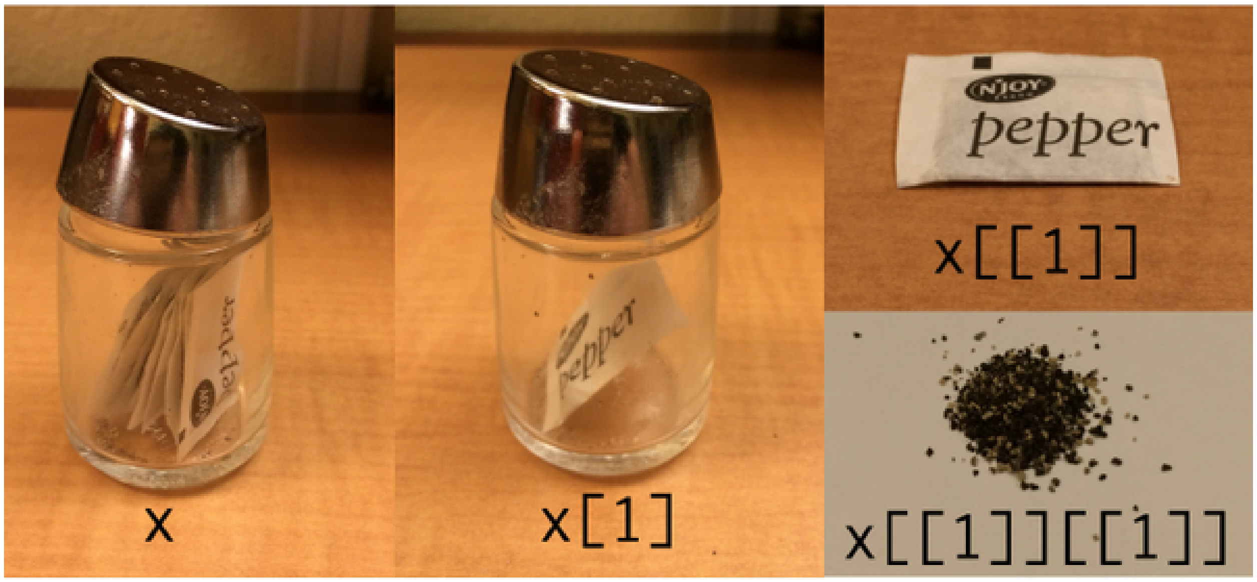


Figure 1: Subsetting a list in R

```
##      [,1] [,2]
## [1,]    4    3
## [2,]    2    7
## [3,]    6    1
```

Naming the columns and rows (always good to have meaningful names)

```
colnames(M) <- c('col1','col2','col3')
rownames(M) <- c('row1','row2')
M
```

```
##      col1 col2 col3
## row1    4    2    6
## row2    3    7    1
```

Q: what happens if the length of the content and the matrix size disagrees?

```
# M <- matrix(1:5, nrow = 2, ncol = 3)
# print(M)
```

Subsetting of a matrix is siimlar to that of a vector, except you need to provide two indices for the two dimensions:

```
M[2,c(1,3)]
```

```
## col1 col3
##    3    1
```

Matrix mutiplication

Dimensions must agree! Vectors are treated by default as column vectors.

```r
# with a vector
M%*%c(1,2,3)
```

```
##      [,1]
## row1   26
## row2   20
```

```r
# with another matrix
M%*%t(M)
```

```
##      row1 row2
## row1   56   32
## row2   32   59
```

Q: what happens if the orientation is misaligned for the vector?

```r
# M%*%t(c(1,2,3))
# What about this?
# t(c(1,2))%*%M
```

Factors

are roughly speaking, storage-wise, an integer vector with labels.

Factors are treated differently from intergers when being operated on. More on this later in the course.

```r
# NA does not count as a factor
factor1 <- factor(c('apple','orange','apple','apple','orange','pear',NA,'orange'))
# coercion to numerical values produces integer values
as.numeric(factor1)
```

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

```r
# read the factor levels
levels(factor1)
```

```
## [1] "apple"  "orange" "pear"
```

```r
# relabel the levels
levels(factor1) <- c("cup","bowl","plate")
# look at the factor again
factor1
```

```
## [1] cup   bowl  cup   cup   bowl  plate <NA>  bowl
## Levels: cup bowl plate
```

Dataframes

Think of dataframes as generalization of matrices, allowing columns to assume different data types.

```r
col1 <- (1:5)/10; col2 <- letters[1:5]; col3 <- round(sin(1:5),3);
```

Numerical objects are converted to characters when concatenating A1 to A3, same as when you create a vector of mixed data types.

```r
A <- cbind(col1 , col2 , col3) # cbind for 'column bind'
print(A)
```

```
##      col1  col2 col3
## [1,] "0.1" "a"  "0.841"
## [2,] "0.2" "b"  "0.909"
## [3,] "0.3" "c"  "0.141"
## [4,] "0.4" "d"  "-0.757"
## [5,] "0.5" "e"  "-0.959"
```

Data frames keep numerical variables numerical, and convert characters to factors.

```
dataframe1 <- data.frame(col1 , col2 , col3)
dataframe1
```

```
##   col1 col2   col3
## 1  0.1    a  0.841
## 2  0.2    b  0.909
## 3  0.3    c  0.141
## 4  0.4    d -0.757
## 5  0.5    e -0.959
```

Viewing dataframe summaries

```
summary(dataframe1)
```

```
##       col1     col2       col3
##  Min.   :0.1   a:1   Min.   :-0.959
##  1st Qu.:0.2   b:1   1st Qu.:-0.757
##  Median :0.3   c:1   Median : 0.141
##  Mean   :0.3   d:1   Mean   : 0.035
##  3rd Qu.:0.4   e:1   3rd Qu.: 0.841
##  Max.   :0.5         Max.   : 0.909
```

Functions

Functions are a great way of reuse and reduce the code you have to write. It wraps a closely-tied chunck of code into a resuable operation with a name. A function is signatured by its name and input. Functions are avaible as a bundle in pre-existing packages, or can be defined as you wish in an R session environment.

How to use an existing function from package

```
# look for manual for a function if you know the function name
?rm
# fuzzy search
??remove
```

Sometimes Google (or your favorarite engine) search is more helpful than "??".

How to define a new function

Use `function()` function

```
f <- function(x){
  x^2-2*x+1
}
```

The last line is returned, you can also use 'return' for clarity

```r
f <- function(x){
  return(x^2-2*x+1)
}
```

Now we can apply it

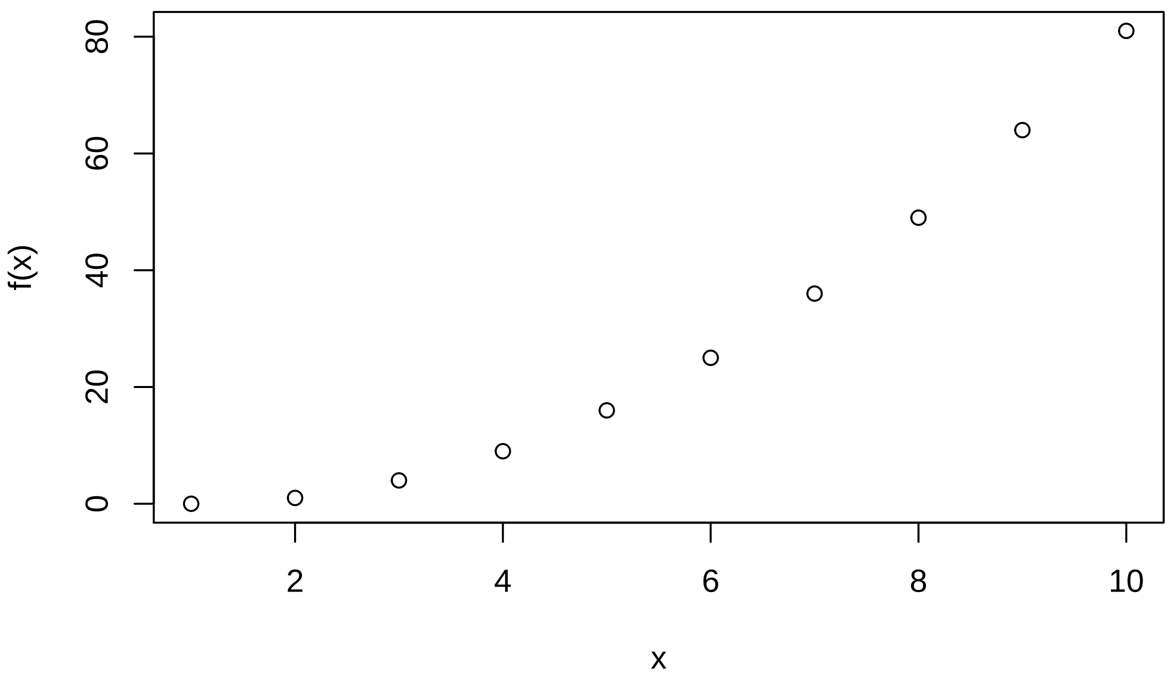
```r
f(x) # works on numericals
```

```
##  [1]  0  1  4  9 16 25 36 49 64 81
```

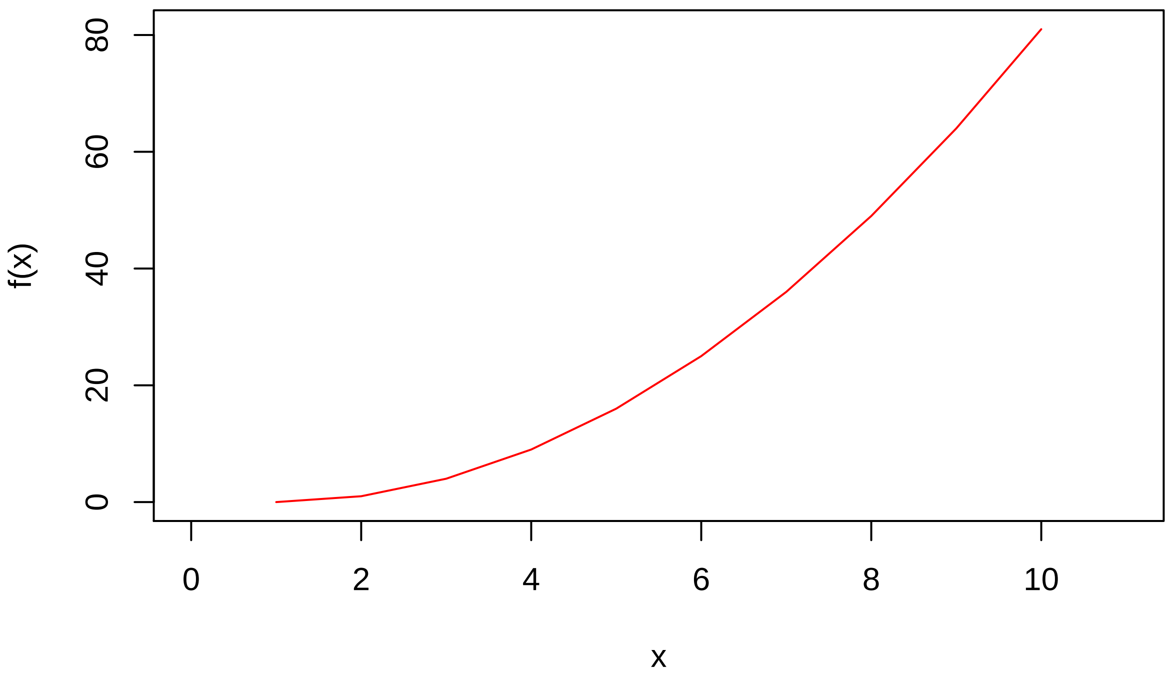
```r
# It contains incompatible operation for strings
# f('test') # won't run
```

Basic plotting

```r
# plot a series against another
plot(x, f(x))
```



```r
# connecting the dots, add colors, plotting ranges
plot(x, f(x) ,type = 'l', col = 'red', xlim = c(0,11))
```



Data wrangling

In real life, most of our time is spent wrangling with datasets; figuring out how things are recorded, dealing with missing data, extract the useful bits of information from a menagerie of variables. Much less time is spent on actual discovery and trying to draw insights from the datasets.

Datasets in classes are mostly curated, clearly annotated. The real world is a lot messier.

The following is an example of real world dataset, still quite well-behaved, but it is closer to what datasets look like out there.

The dataset is adapted from:

Allison, T. and Chichetti, D. (1976) Sleep in mammals: ecological and constitutional correlates. Science 194 (4266), 732–734.

- The dataset consists of animal sleep data with 62 observations on 10 variables.
- Some of the variables are missing.
- The dataset is splitted into two (deliberately): First contains the observation ID plus the first 6 variables. Second contains the observation ID plus the last 4 variables.

We are going to read the datasets, combine them according to their ID, calculate some simple statistics, and visulaize the results.

Read the dataset into R

Download the datasets sleep1.csv, sleep2.csv here by following the links or download from Canvas.

Inspect the datasets. Read both files into R.

You should be getting the following header names and data types

```
sleepdata1 <- read.csv(file = "./data/sleep1.csv", header = T,na.strings = "NULL")
sleepdata2 <- read.csv(file = "./data/sleep2.csv", header = T,na.strings = "NULL")

sapply(sleepdata1,class)
```

```
##        ID   BodyWgt  BrainWgt      NonD     Dream     Sleep      Span
## "integer" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
```

```
sapply(sleepdata2,class)
```

```
##        ID      Gest      Pred       Exp    Danger
## "integer" "numeric" "integer" "integer" "integer"
```

Combine the datasets according to observation ID

We shoud be merging the two datasets where their ID's match.

The combined headers and data types look like this:

```
sleepdata <- merge(sleepdata1,sleepdata2)
sapply(sleepdata, class)
```

```
##        ID   BodyWgt  BrainWgt      NonD     Dream     Sleep      Span
## "integer" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
##      Gest      Pred       Exp    Danger
## "numeric" "integer" "integer" "integer"
```

Q: Does `cbind` work? Why? What about `merge`?

Calculate the proportion of dream time in sleep

Notice there is a numeric column "Dream" and a numerical column "Sleep". We want to calculate the ratio of the two, where neither of them are `NA`'s.

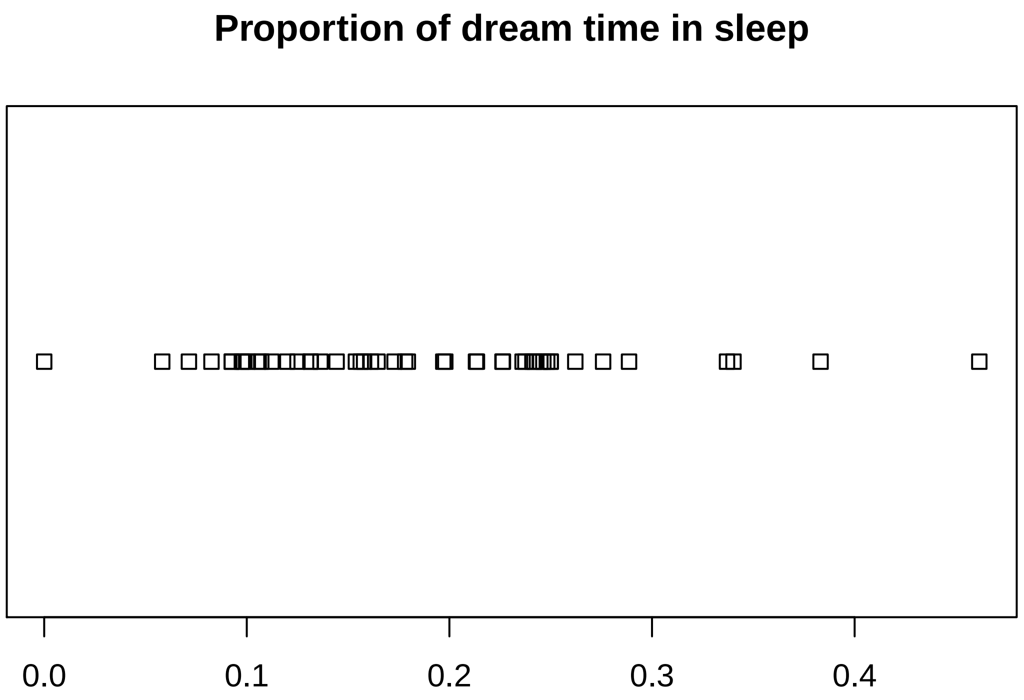
Then append the new column to the dataframe. Call the new Column "Ratio". The first 10 rows look like this:

```
sleepdata <- cbind(sleepdata,"Ratio" = sleepdata$Dream/sleepdata$Sleep)
head(sleepdata[2:12],10)
```

```
##     BodyWgt BrainWgt NonD Dream Sleep Span Gest Pred Exp Danger
## 1  6654.000   5712.0   NA    NA   3.3 38.6  645    3   5      3
## 2     1.000      6.6  6.3   2.0   8.3  4.5   42    3   1      3
## 3     3.385     44.5   NA    NA  12.5 14.0   60    1   1      1
## 4     0.920      5.7   NA    NA  16.5   NA   25    5   2      3
## 5  2547.000   4603.0  2.1   1.8   3.9 69.0  624    3   5      4
## 6    10.550    179.5  9.1   0.7   9.8 27.0  180    4   4      4
## 7     0.023      0.3 15.8   3.9  19.7 19.0   35    1   1      1
## 8   160.000    169.0  5.2   1.0   6.2 30.4  392    4   5      4
## 9     3.300     25.6 10.9   3.6  14.5 28.0   63    1   2      1
## 10   52.160    440.0  8.3   1.4   9.7 50.0  230    1   1      1
##          Ratio
## 1           NA
## 2  0.240963855
## 3           NA
## 4           NA
## 5  0.461538462
## 6  0.071428571
## 7  0.197969543
## 8  0.161290323
## 9  0.248275862
## 10 0.144329897
```

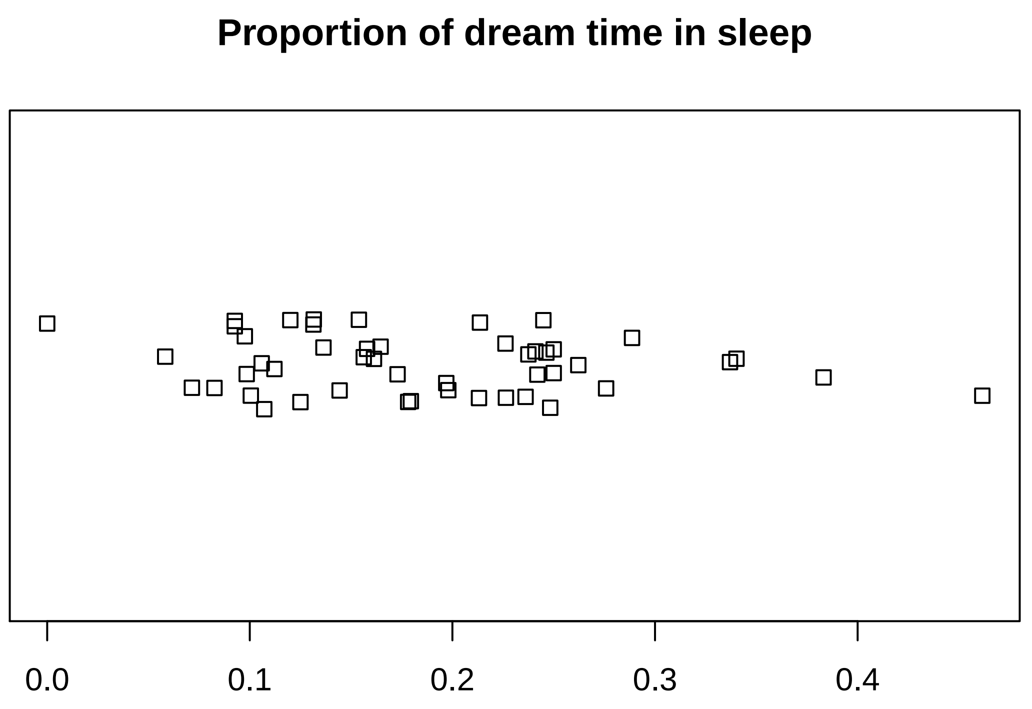
Stripchart

```
stripchart(sleepdata$Ratio,
           main =  "Proportion of dream time in sleep")
```



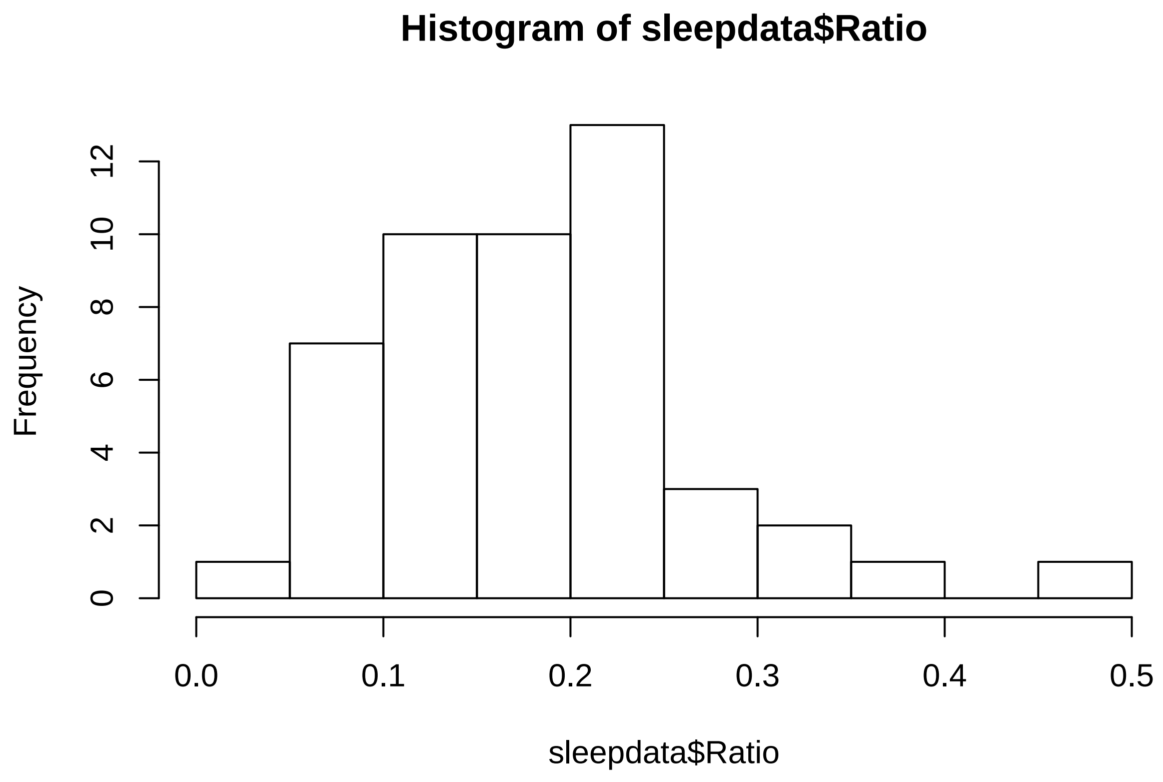
It is a little more pleasing to the eye to see a jittered stripchart.

```
stripchart(sleepdata$Ratio, method = 'jitter',
           main =  "Proportion of dream time in sleep")
```



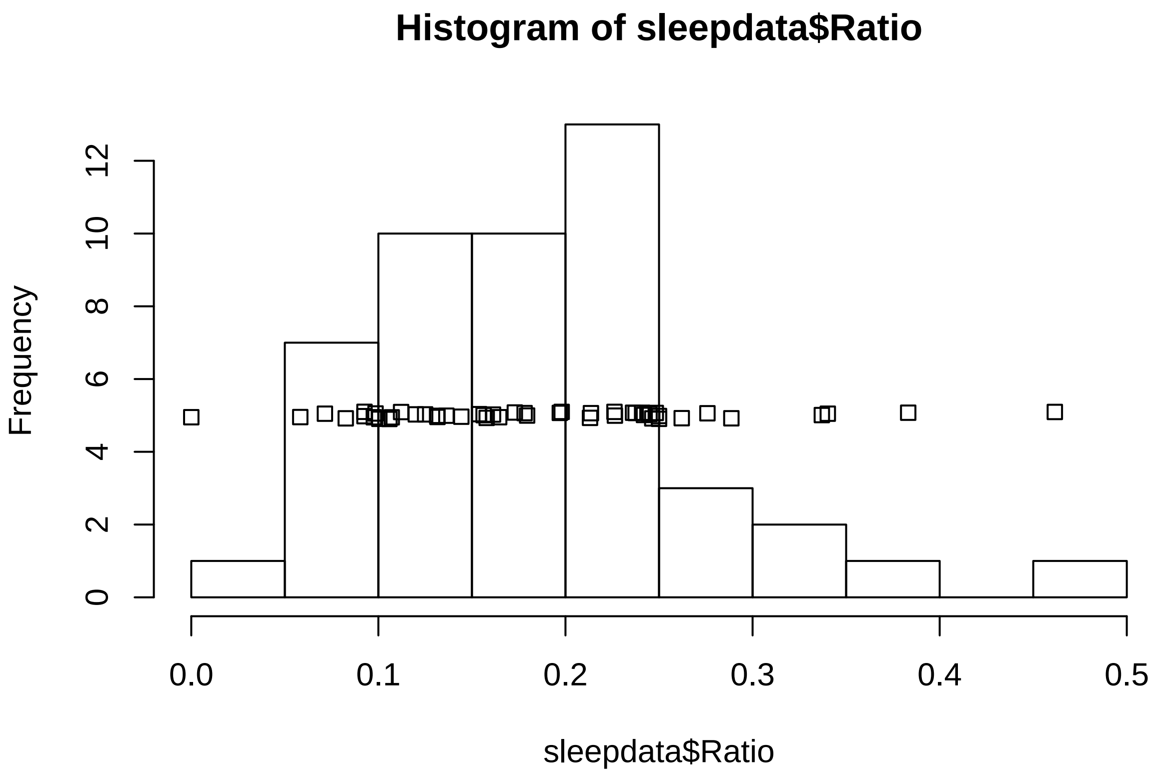
Histogram

```r
hist(sleepdata$Ratio)
```



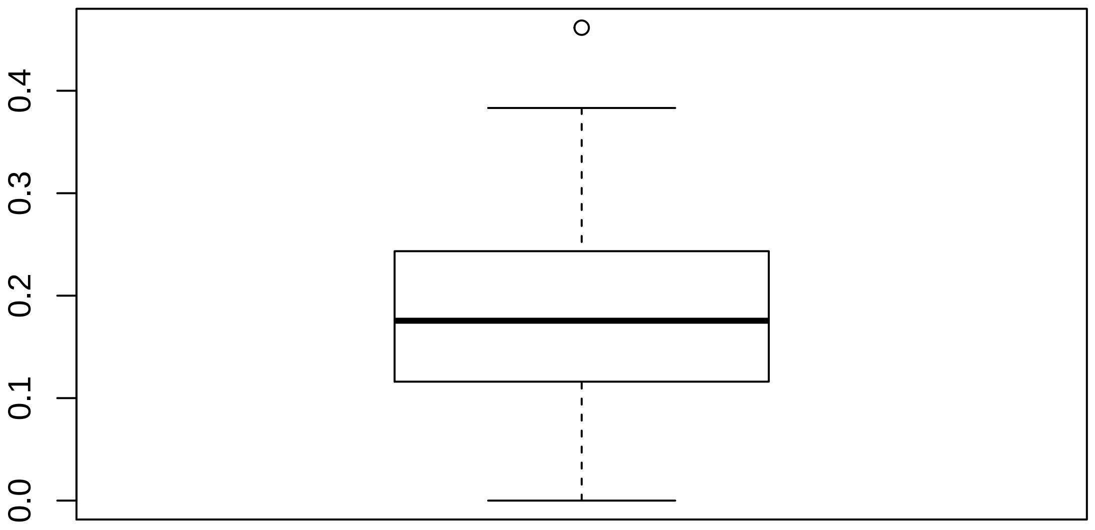
You can also overlay a stripchart over the histogram.

```r
hist(sleepdata$Ratio)
stripchart(sleepdata$Ratio, method = 'jitter', add=TRUE, at = 5)
```



Boxplot

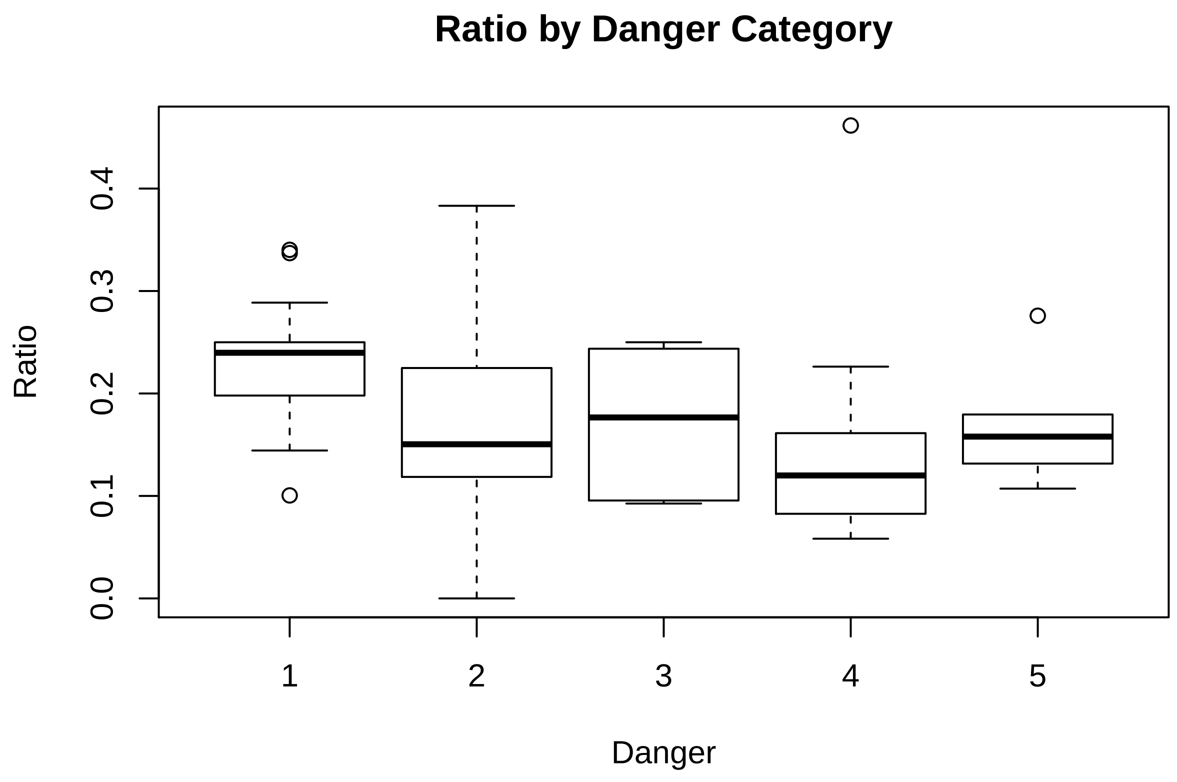
```r
boxplot(sleepdata$Ratio)
```



Boxplot of a variable against another

Now we want to viualize the relationship between the variables "Ratio" and "Danger" using a boxplot.

```
with(sleepdata,boxplot(Ratio~Danger,
                       xlab='Danger',
                       ylab = 'Ratio',
                       main='Ratio by Danger Category'))
```



Tip: The function `with` can shorten the code significantly for expressions involving dataframes.

`with` creates an environment from the dataframe and places it in the search path, so that when you refer to the variables in the dataset, you no longer have refer to the dataset by name.

For example instead of writing

```
plot(really_long_dataframe_name$really_long_variable_name_1,
     really_long_dataframe_name$really_long_variable_name_2)
```

you can write

```
with(really_long_dataframe_name,
     plot(really_long_variable_name_1,really_long_variable_name_2))
```

See more on `?with`.

Order the dataset by a variable

What if we want to reorder the dataframe accoring to the variable "Ratio", in increasing order?

```
sleepdata <- sleepdata[order(sleepdata$Ratio),]
head(sleepdata[,2:12],10)
```

```
##    BodyWgt BrainWgt NonD Dream Sleep Span Gest Pred Exp Danger       Ratio
## 16   3.000    25.00  8.6   0.0   8.6 50.0   28    2   2      2 0.000000000
## 60   4.190    58.00  9.7   0.6  10.3 24.0  210    4   3      4 0.058252427
## 6   10.550   179.50  9.1   0.7   9.8 27.0  180    4   4       4 0.071428571
## 43  10.000   115.00 10.0   0.9  10.9 20.2  170    4   4       4 0.082568807
## 52   3.600    21.00  4.9   0.5   5.4  6.0  225    3   2       3 0.092592593
## 58   2.000    12.30  4.9   0.5   5.4  7.5  200    3   1       3 0.092592593
## 28   1.040     5.50  7.4   0.8   8.2  7.6   68    5   3       4 0.097560976
## 37   0.023     0.40 11.9   1.3  13.2  3.2   19    4   1       3 0.098484848
## 33   0.010     0.25 17.9   2.0  19.9 24.0   50    1   1       1 0.100502513
## 42   0.480    15.50 15.2   1.8  17.0 12.0  140    2   2       2 0.105882353
```

Hint: Does `sort` do? What about `order`?

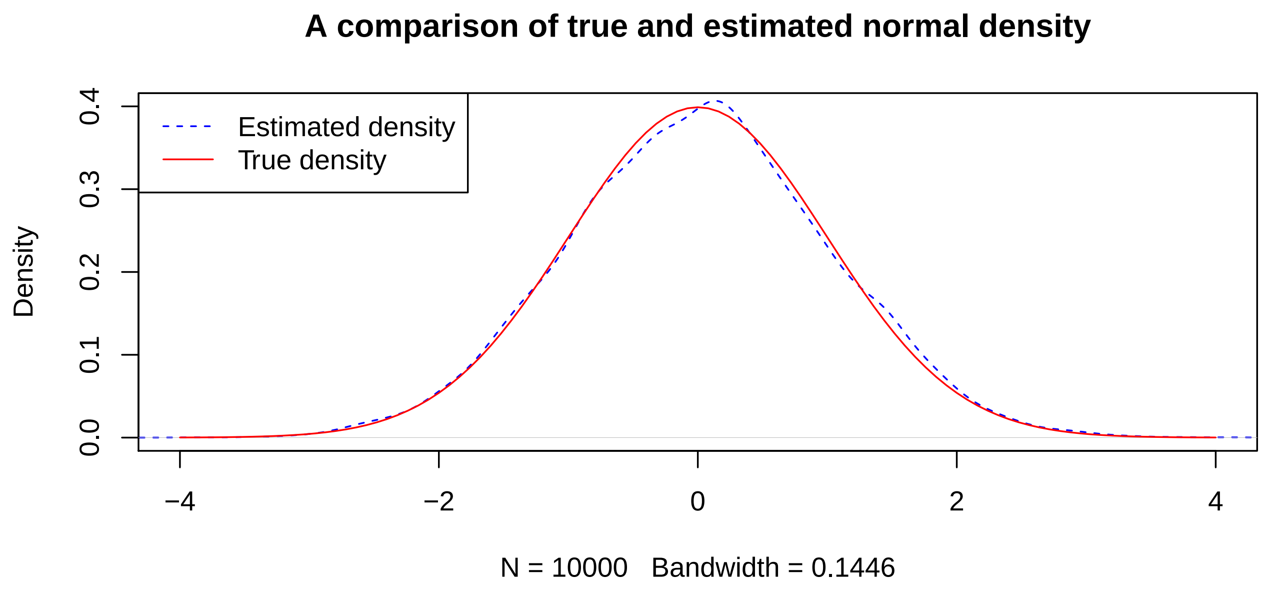
More on plotting

There are many ways you can make your plots for

This example contains a few frequently used plotting parameters:

- Line/Point styles: `type`, `lty`, `pch`, `col`, …
  - `type`: consult `?plot`, it refers to the type of the line. 'b' for showing both the points and the line, 'o' for both but over-positioning each other, 'l' for line-only and 'p' for points-only.
  - `lty`: line type, take integer values.
  - `pch`: the point marker used, take integer or character values.
  - `col`: color, take integer or character values.
- Plot range: `xlim`, `ylim`. Here we expanded the plot ranges to have more comfortable margins.
- Texts: `main`, `sub`, `xlab`, `ylab`, …
  - `main`: main title of the plot.
  - `sub`: subtitle.
  - `xlim`, `ylim`: ranges of axes.
- Font sizes: `cex` (default), `cex.main`, `cex.sub`, `cex.lab`, `cex.axis`, …
- Axes: `axes` …
- Margin: `par(mar = )` …
- Layout: `layout`, `par(mfrow = )` …

```r
simulated.X <- rnorm(n=10000);
dens <- density(simulated.X);
plot(
  dens, col = 'blue', lty = 2, xlim = c(-4,4), ylim = c(0,0.4),
  main = "A comparison of true and estimated normal density"
)
par(new = T)
curve(
  dnorm,lty = 1, col = 'red',xlim = c(-4,4), ylim = c(0,0.4),
  axes = FALSE, xlab = "", ylab = ""
)
legend(
  x = 'topleft',legend = c('Estimated density','True density'),
  col = c('blue','red'), lty = c(2,1)
)
```



It is possible to come up with pretty plots using only base packages. Plotting needs a little practice (and some Googling).

Extras

- It is always good to keep in mind that computers are finite machines.
- They are only approxiamtely accurate.
- Your algorithm matters.

E.g.

```
options(digits = 22)
sum(1/1:500000)
```

```
## [1] 13.69958004230552717218
```

```
sum(1/500000:1)
```

```
## [1] 13.69958004230552894853
```

Results are close (for all practical purposes), but bot identical.

Advanced

Advanced R is a good reference if you are into more details of R.