R intro

PART I: R Basics

Basic math

```
2+2
```

```
## [1] 4
```

```
1/3  # R as standard has more digits than it shows
```

```
## [1] 0.3333333
```

```
print(1/3, digits = 12)
```

```
## [1] 0.333333333333
```

```
sqrt(2)
```

```
## [1] 1.414214
```

Saving the results

```
a <- 34+45  # can also use = instead of <-
a
```

```
## [1] 79
```

Vectors

```
c(1, 5, -4) # c means combine/concatenate
```

```
## [1]  1  5 -4
```

```
v <- c(2, 3, -5); v  # several commands on one line are separated with semi-colon
```

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

```
w <- 1:3; w  # colon is used for sequences of numbers
```

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

```r
w <- seq(1, 5, by = 2) # `seq` is used for more general regular sequences
v+w  # adding vectors
```

```
## [1] 3 6 0
```

```r
v[2]  # extracting element 2 of v
```

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

```r
v[2:3] # extracting element 2 and 3 of v
```

```
## [1]  3 -5
```

```r
c(v, w) # easy to join several vectors into a long vector
```

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

Matrices

```r
A <- matrix(1:9, 3, 3); A  # written by column
```

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

```r
B <- matrix(1:9, 3, 3, byrow=TRUE); B  # written by row
```

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

```r
C <- cbind(v, w); # bind vectors v and w as columns of a matrix
```

Manipulating elements

```r
A[2,2]  # entry [2,2]
```

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

```r
A[,2]  # column 2
```

```
## [1] 4 5 6
```

```
A[2,]  # row 2
```

```
## [1] 2 5 8
```

```
A[3,3] <- 0 # changing element at entry [3,3] in a matrix
A
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    0
```

All operations on vectors and matrices are elementswise

```
A+B  # fine for matrix sum
```

```
##      [,1] [,2] [,3]
## [1,]    2    6   10
## [2,]    6   10   14
## [3,]   10   14    9
```

```
A*B  # not a proper matrix product - elementwise product
```

```
##      [,1] [,2] [,3]
## [1,]    1    8   21
## [2,]    8   25   48
## [3,]   21   48    0
```

```
A^{-1}  # not proper matrix inverse - elementwise inverse
```

```
##           [,1]      [,2]      [,3]
## [1,] 1.0000000 0.2500000 0.1428571
## [2,] 0.5000000 0.2000000 0.1250000
## [3,] 0.3333333 0.1666667       Inf
```

Help

All built-in functions have help pages accessible by `?`

```
?matrix
```

If you don't know the name of the command, Google is your friend

Data frames

Data frames look like matrices and they are the default data format in R

```r
data.frame(1:4, 5:2, c(1,3,4,7))  # vectors get a column each
```

```
##   X1.4 X5.2 c.1..3..4..7.
## 1    1    5             1
## 2    2    4             3
## 3    3    3             4
## 4    4    2             7
```

```r
y <- data.frame(height = c(178, 182, 171), weight = c(72, 76, 71))  # the columns can be given meaningf
y
```

```
##   height weight
## 1    178     72
## 2    182     76
## 3    171     71
```

```r
y[,2]  # same extraction notation as matrices
```

```
## [1] 72 76 71
```

```r
y$weight  # or use $ with column name
```

```
## [1] 72 76 71
```

```r
str(y) # Gives the structure of any R object. Very useful for complicated objects.
```

```
## 'data.frame':    3 obs. of  2 variables:
##  $ height: num  178 182 171
##  $ weight: num  72 76 71
```

importing data using `read.table()`/`read.csv()` No header is assumed by default in `read.table()`

```r
beetles_url <- "https://asta.math.aau.dk/course/bayes/2021/?file=./beetles.dat"
beetles <- read.table(beetles_url)
head(beetles)
```

```
##       V1 V2
## 1 1.6907  0
## 2 1.6907  0
## 3 1.6907  0
## 4 1.6907  0
## 5 1.6907  0
## 6 1.6907  0
```

First line is assumed to be header by default in `read.csv()`

```r
newcomb_url <- "https://asta.math.aau.dk/course/bayes/2021/?file=./newcomb.csv"
newcomb <- read.csv(newcomb_url)
head(newcomb)
```

```
##   day   time
## 1   1 24.828
## 2   1 24.826
## 3   1 24.833
## 4   1 24.824
## 5   1 24.834
## 6   1 24.756
```

The `class` function is useful for figuring out what you are working with

```
class(newcomb)  # a data frame
```

```
## [1] "data.frame"
```

```
class(newcomb[,1])  # a integer vector (almost the same as numeric in R)
```

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

```
class(newcomb[,2])  # a vector of floating point numbers (numeric)
```

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

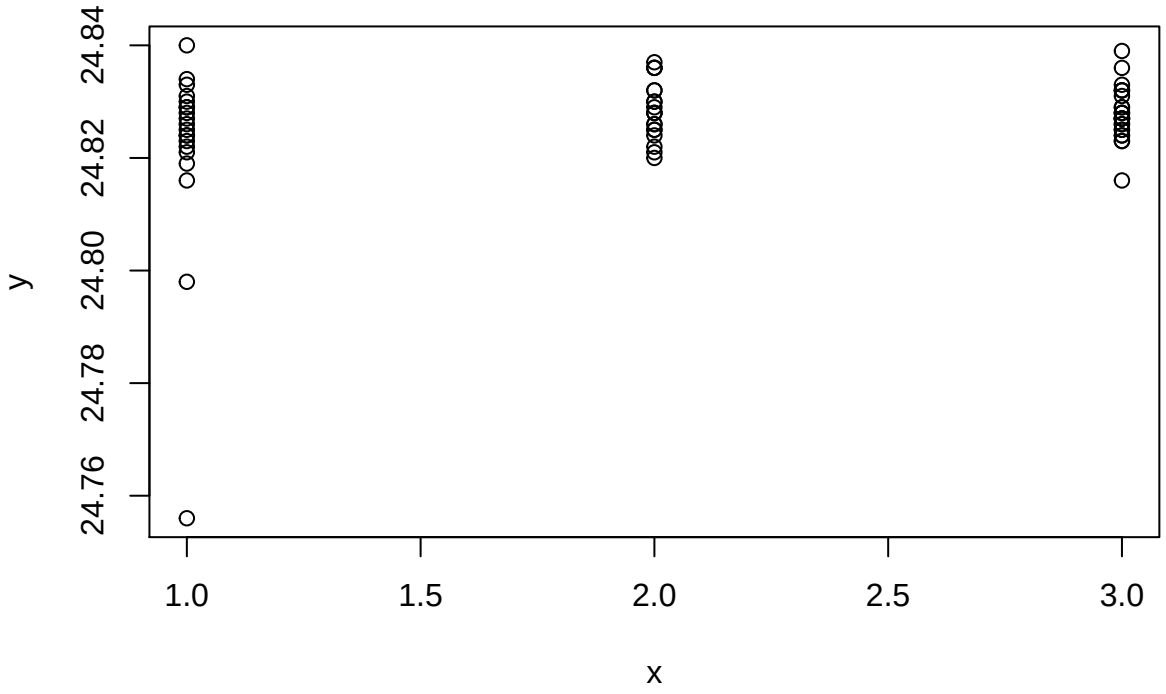
```
str(newcomb) # Again, this is the best way to see all this in one go
```

```
## 'data.frame':    66 obs. of  2 variables:
##  $ day : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ time: num  24.8 24.8 24.8 24.8 24.8 ...
```

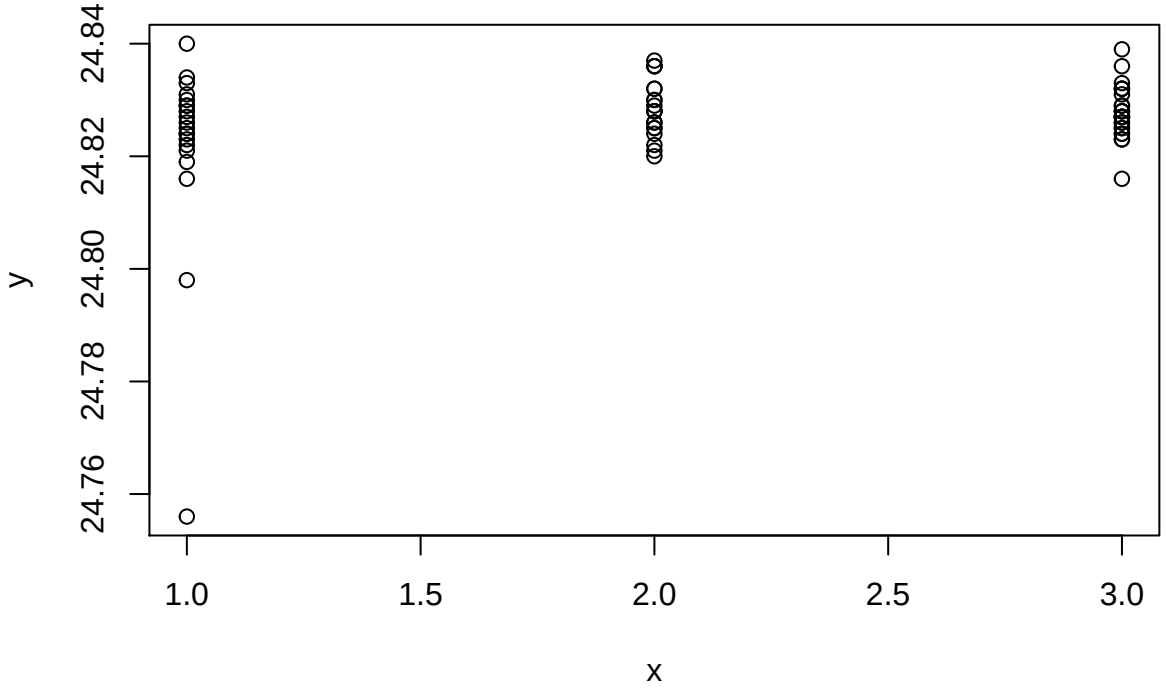
Plotting

The generic plot function

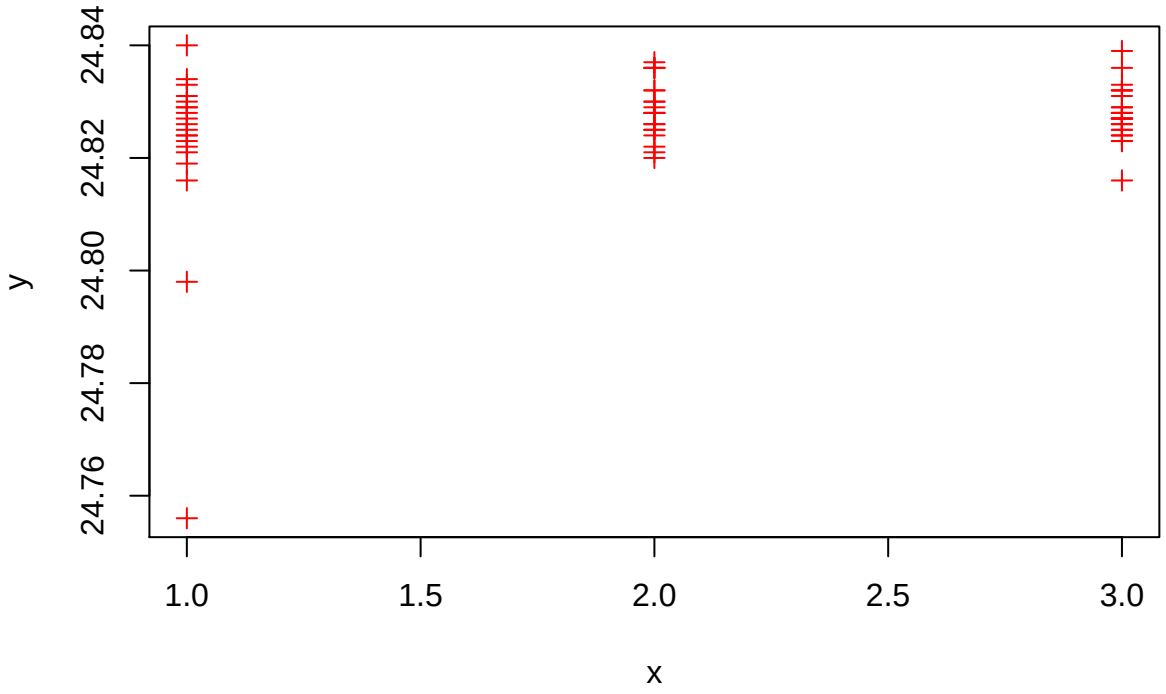
```
x <- newcomb$day
y <- newcomb$time
plot(x, y)
```



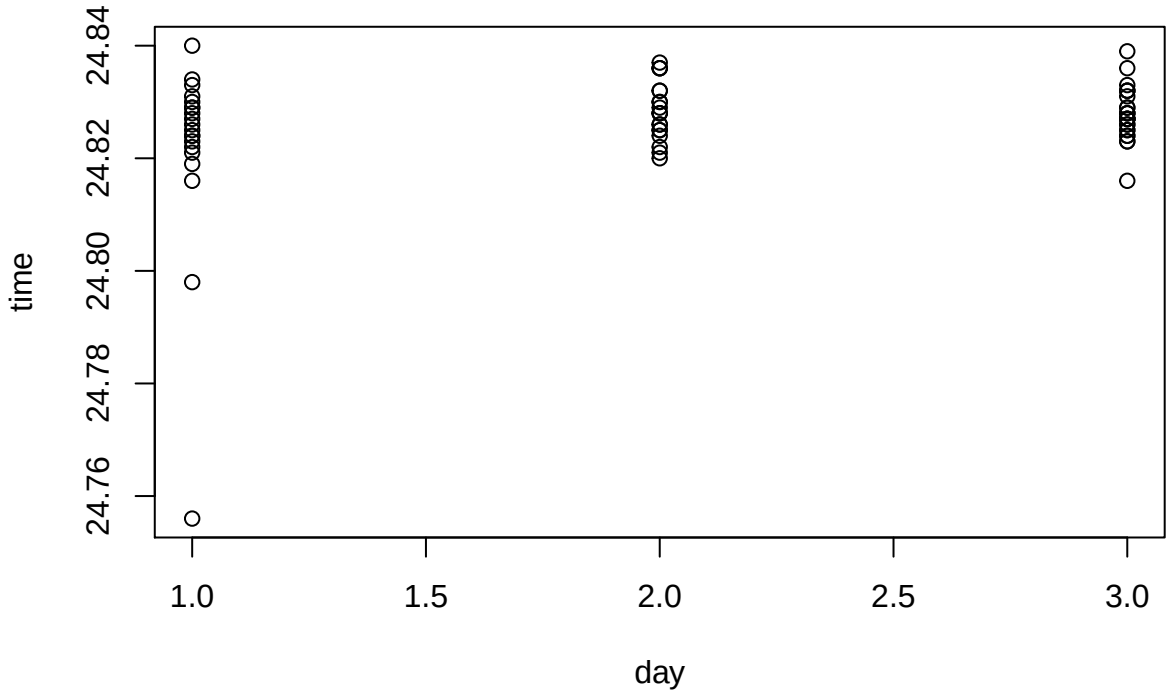

```
plot(y ~ x)  # the same, ~ means "as function of", note different order
```




```
plot(x, y, pch=3, col="red")  # many things can be controlled in plot
```



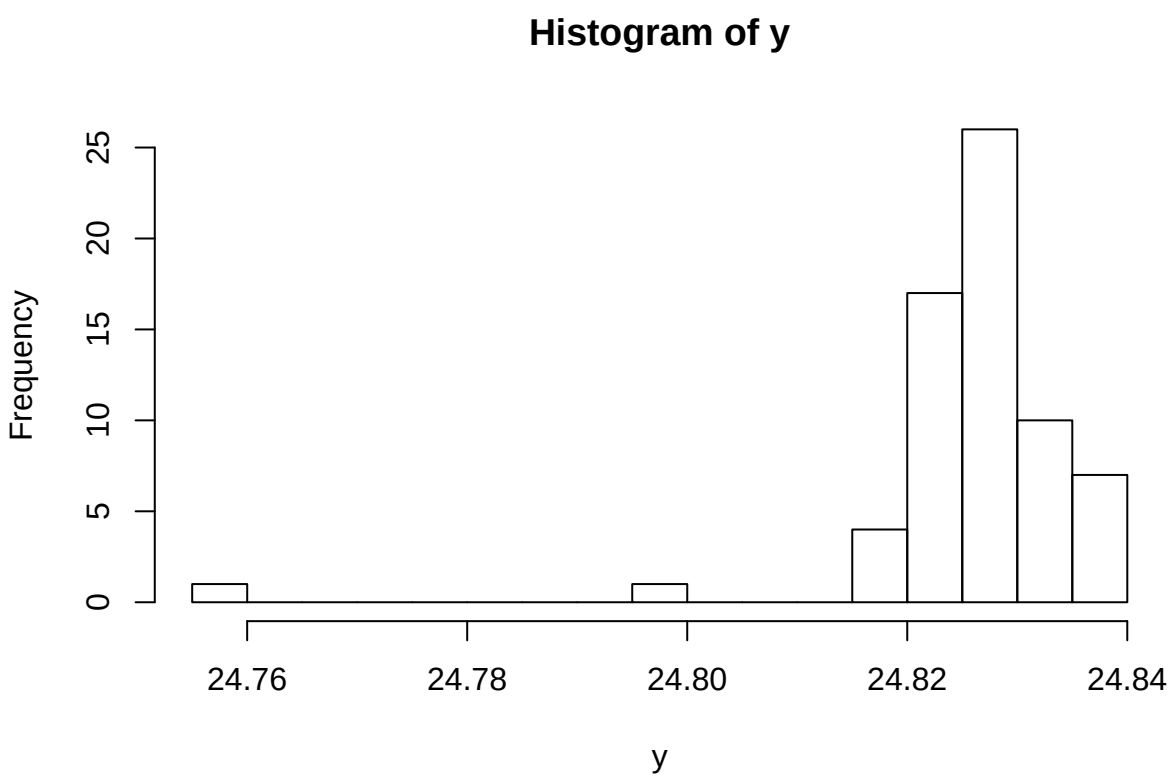

```
plot(newcomb)  # complex objects can also be put into plot - what it does depends on the object
```



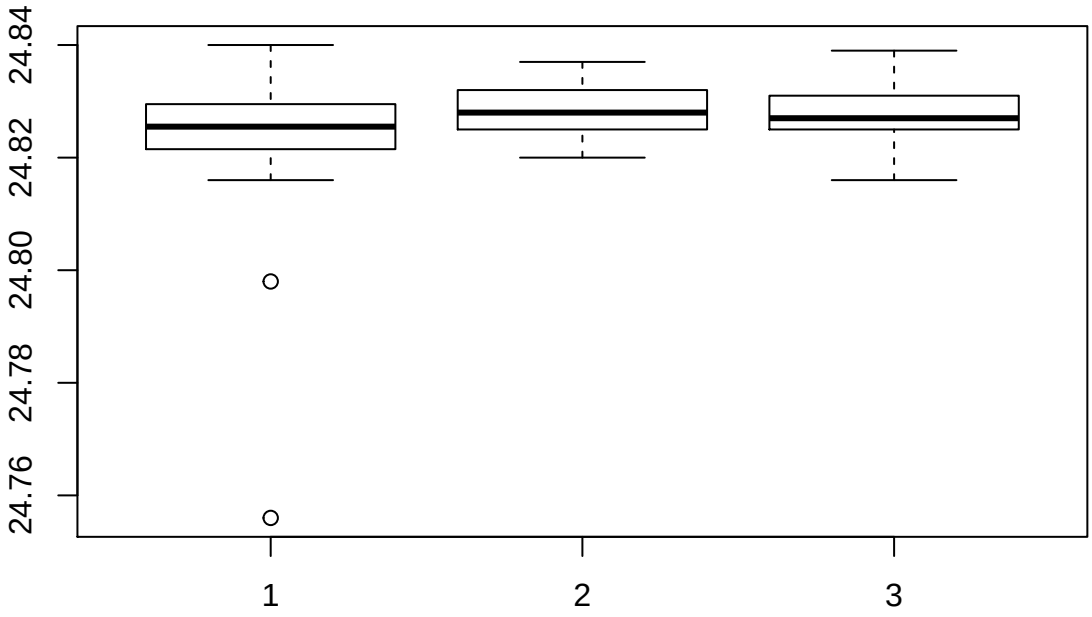
Note: `plot` is a generic function, that under the hood dispatches to other functions e.g. `plot.default` or `plot.data.frame`.
Even though you only write `plot(...)` you may still need to look at the help for these other functions

Graphical data summaries

```
hist(y, breaks=20)  # good for getting an overview of data
```




```
boxplot(y ~ x)  # good for comparing data
```



Numerical data-summaries

```
mean(y)  # average
```

```
## [1] 24.82621
```

```
median(y)  # median
```

```
## [1] 24.827
```

```
var(y)  # variance
```

```
## [1] 0.000115462
```

```
sd(y)  # standard deviation
```

```
## [1] 0.01074532
```

```
range(y)  # range
```

```
## [1] 24.756 24.840
```

```
quantile(y)  # quantiles
```

```
##       0%      25%      50%      75%     100%
## 24.75600 24.82400 24.82700 24.83075 24.84000
```

```
summary(y)  # can be used for most objects in R
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   24.76   24.82   24.83   24.83   24.83   24.84
```

PART II: Distributions in R

R can be used for calculating density functions and distribution functions of most known distributions

```
dnorm(1, mean=0, sd=1)  # density function for normal dist. with mean 0 and sd 1 evaluated at 1
```

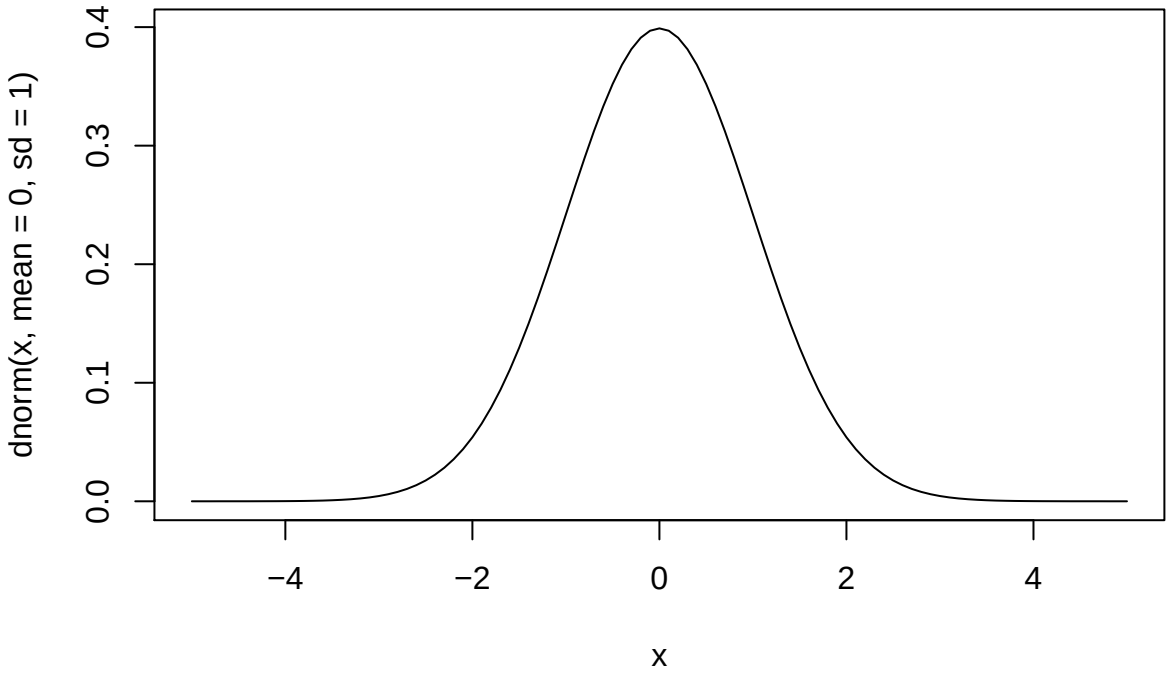
```
## [1] 0.2419707
```

```
pnorm(1, mean=0, sd=1)  # Corresponding distribution function
```

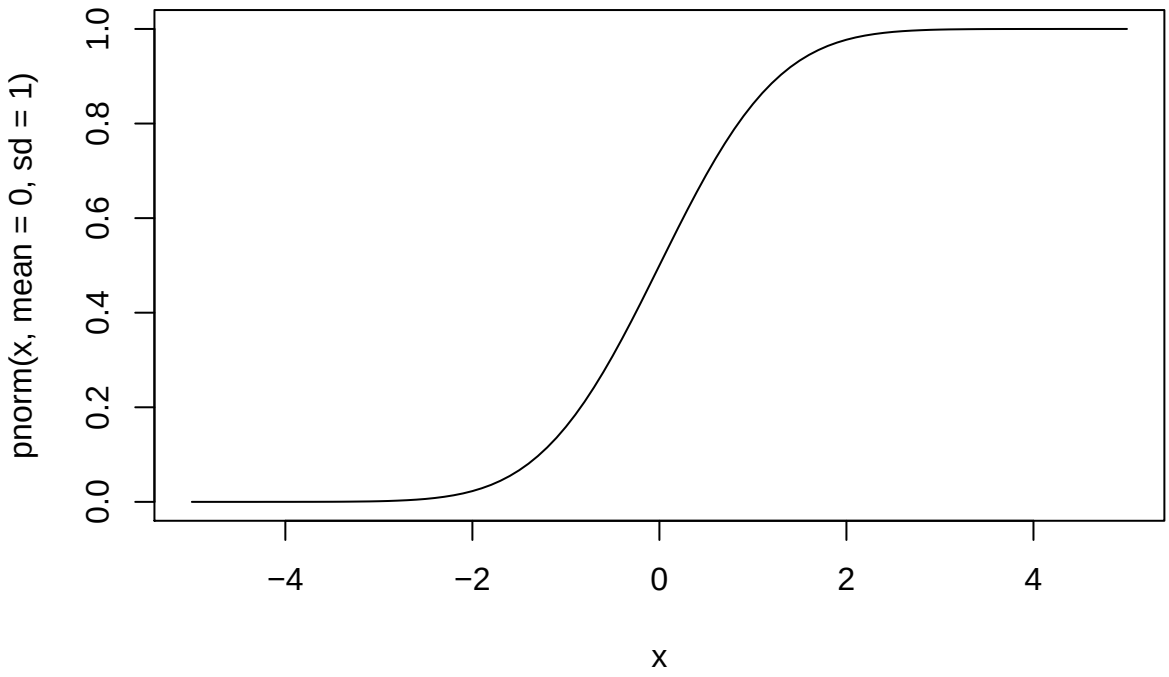
```
## [1] 0.8413447
```

let's plot them - the `curve` function is useful for plotting mathematical functions in R

```r
curve(dnorm(x, mean=0, sd=1), from=-5, to=5)  # x must be called x in curve
```




```r
curve(pnorm(x, mean=0, sd=1), from=-5, to=5)
```



the inverse distribution function (quantile function)

```r
qnorm(0.8, mean=0, sd=1)
```

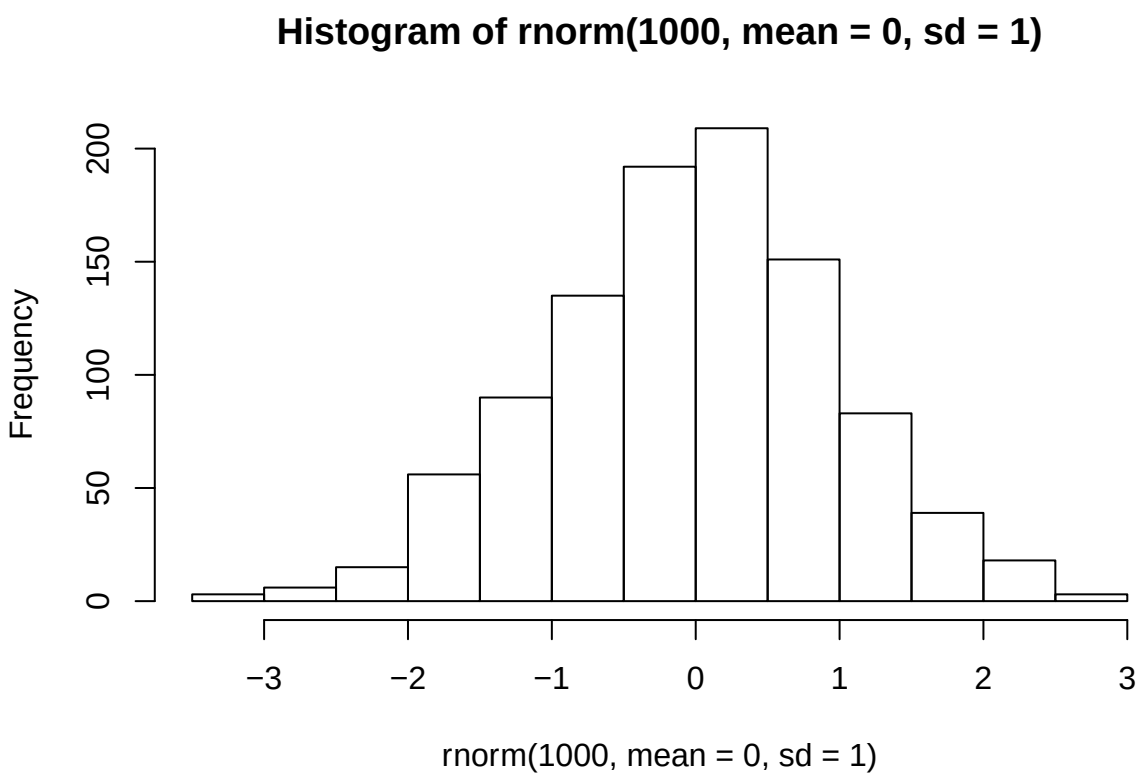
```
## [1] 0.8416212
```

R can also simulate the distributions

```
rnorm(5, mean=0, sd=1)  # 5 simulations
```

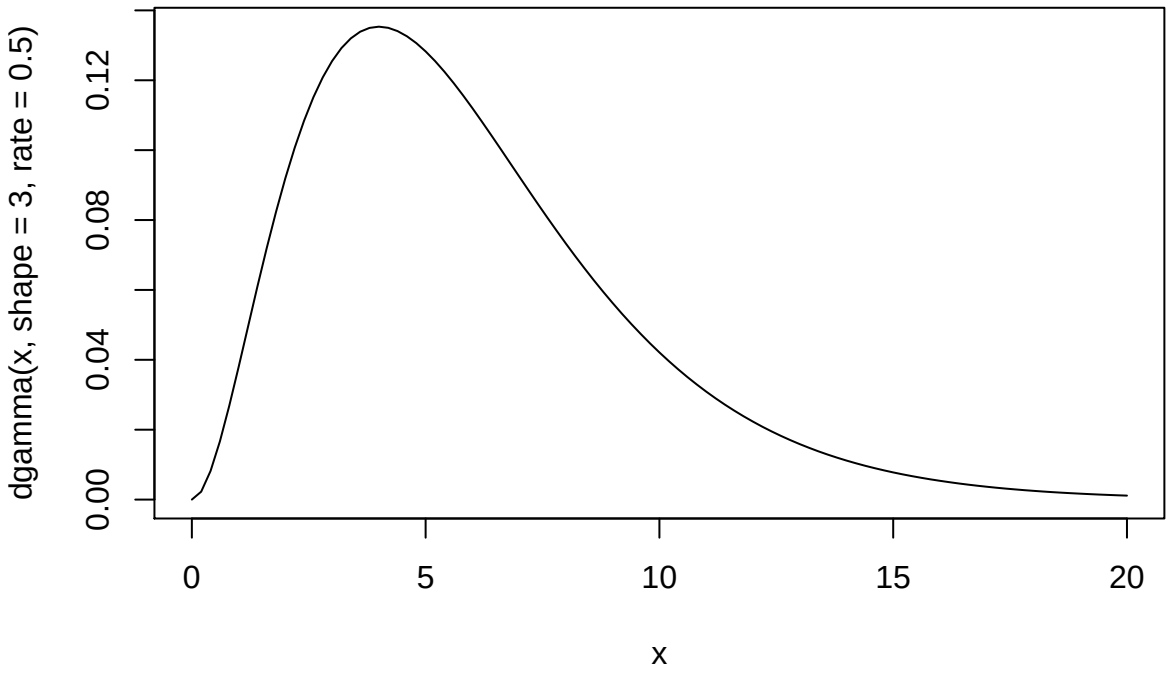
```
## [1]  0.3253555 -0.3702653  0.9546970 -2.5823757  0.1253694
```

```
hist(rnorm(1000, mean=0, sd=1))  # histogram of 1000 simulations
```



Functions for handling distributions come with the following naming convention:
First part:
`d` = **D**ensity function
`p` = distribution function (cumulative **P**robability)
`q` = inverse distribution function (**Q**uantile)
`r` = simulate (**R**andom number generator)
Last part: distribution name, e.g:
`norm` = normal
`exp` = exponential
`gamma` = gamma
`beta` = beta
`binom` = binomial
etc.

```
curve(dgamma(x, shape = 3, rate = 0.5), from=0, to=20)
```



Multivariate distribution – example of a package

If R does not contain the functions/statistics you need, odds are that somebody has implemented it in a package. Installing a package - (here a package for multivariate t and normal distribution)

```
install.packages("mvtnorm")  # only need to install it once
```

Loading a package

```
library(mvtnorm)  # need to do this everytime R is started
```

In Rstudio you can also use the Packages tab (lower right panel by default).

Now the following functions work for calculating the density of and simulating multivariate normal distributions

```
dmvnorm(c(1,2,2), mean = rep(0,3), sigma = diag(3))  # evaluating the 3-dim standard normal density
```

```
## [1] 0.0007053506
```

```
rmvnorm(5, mean = rep(0,3), sigma = diag(3)) # every row is a simulation
```

```
##            [,1]       [,2]       [,3]
## [1,]  1.1566176  0.5338822 -0.7475116
## [2,]  1.4095179 -0.2718381 -0.9203554
## [3,] -0.9922887  0.7492173  1.0175581
## [4,]  0.4861117  0.4689197  0.2792520
## [5,]  0.9568027 -1.7170356  0.7970160
```

If you don't know what a package contains, you can try

```
library(help = "mvtnorm")
```

It is possible to use a function from a package without loading the package first via `library`. This is done using the :: notation like this:

```
mvtnorm::rmvnorm(5, mean = rep(0,3), sigma = diag(3))
```

```
##             [,1]        [,2]       [,3]
## [1,] -0.1364281 -0.94738220  0.2365588
## [2,]  2.4380129 -0.59428624  1.2161294
## [3,] -1.9643519  0.13117526 -2.0195807
## [4,] -0.1484876  0.16686179 -0.5524915
## [5,]  1.2366506  0.07183109 -0.1858097
```

There are thousands of other packages for specific needs.

PART III: Programming – functions, loops, etc.

For-loop

Calculating 1+2+...+10 as an example

```
s <- 0
for (i in 1:10){
  s <- s + i
}  # any vector can be used instead of 1:10
s
```

```
## [1] 55
```

Calculating the first ten Fibonacci numbers

```
f <- rep(0,10)
f[1] <- f[2] <- 1
for (i in 3:10){
  f[i] <- f[i-2]+f[i-1]
}
f
```

```
##  [1]  1  1  2  3  5  8 13 21 34 55
```

Note that built-in functions are usually faster than for-loops created from scratch

If-then-else conditions

Determining the sign of a number

```
x <- -3
if (x<0) {
  signx <- -1
} else{
  if (x==0){
    signx <- 0
  } else{
    signx <- 1
  }
}
signx
```

```
## [1] -1
```

Functions

A function for finding the sign of a number

```
signfct <- function(x){
  # syntax: fct_name <- function(input1, input2, ...){blablabla}
  signx <- 0 # Assume 0 until found otherwise
  if (x<0) {
    signx <- -1
  }
  if (x>0){
    signx <- 1
  }
  return(signx)
}
signfct(-3); signfct(0); signfct(0.2)
```

```
## [1] -1
```

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

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

There is a built-in function `sign`

```
sign(-3); sign(0); sign(0.2)
```

```
## [1] -1
```

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

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

```
sign(-3:4)  # this will even take vectors or matrices
```

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

```r
signfct(-3:4)  # our function is not that smart, since `if` only accepts a single value
```

```
## Warning in if (x < 0) {: the condition has length > 1 and only the first
## element will be used

## Warning in if (x > 0) {: the condition has length > 1 and only the first
## element will be used

## [1] -1
```

Morale: always think about all the types of input you would like to have and try them out.