STAT 210
Applied Statistics and Data Analysis
Basic Functions

Joaquín Ortega

Fall 2020

Basic Functions

c

The concatenation function `c` combines elements to create a vector. The elements are listed in order and separated by a comma:

```
x <-c(1,1.5,2,2.5)
x
```

```
## [1] 1.0 1.5 2.0 2.5
```

A variable can also be used as an argument for this function. For instance, if we want to add the number 3 to the vector `x` we can write:

```
(x <- c(x,3))
```

```
## [1] 1.0 1.5 2.0 2.5 3.0
```

c

and also characters can be used:

```
(y <- c('this','is','an','example'))
```

```
## [1] "this"    "is"      "an"      "example"
```

However, if one combines numbers and characters, all entries of the vector will be considered as characters and it is not possible to operate on them as numbers:

```
(z <- c(x,'a'))
```

```
## [1] "1"   "1.5" "2"   "2.5" "3"   "a"
```

If you want to avoid the quotation marks when printing characters, you may use the function `cat`:

```
cat(z)
```

```
## 1 1.5 2 2.5 3 a
```

seq

The function seq is used to form regular sequences of numbers. The basic syntax is

```
seq(from=1, to=1, by=((to-from)/length.out-1),
    length.out = NULL)
```

Let's see some examples:

```
seq(0,100,5)
```

```
##  [1]   0   5  10  15  20  25  30  35  40  45  50
## [12]  55  60  65  70  75  80  85  90  95 100
```

```
seq(1955,1966,1)
```

```
##  [1] 1955 1956 1957 1958 1959 1960 1961 1962 1963
## [10] 1964 1965 1966
```

```
seq(10,12,0.2)
```

```
##  [1] 10.0 10.2 10.4 10.6 10.8 11.0 11.2 11.4 11.6
## [10] 11.8 12.0
```

seq

1 is the default value for the lower limit of the sequence and also for the increment:

```
seq(1955,1966)
```

```
##  [1] 1955 1956 1957 1958 1959 1960 1961 1962 1963
## [10] 1964 1965 1966
```

```
seq(5)
```

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

```
seq(5,-1)
```

```
## [1]  5  4  3  2  1  0 -1
```

seq

There is a shorthand for this function when the increment is 1:

```
1:5
```

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

```
5:1
```

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

```
50:60/5
```

```
##  [1] 10.0 10.2 10.4 10.6 10.8 11.0 11.2 11.4 11.6
## [10] 11.8 12.0
```

rep

The function rep replicates a pattern. The syntax is

```r
rep(x,times)
```

where x is the object to be replicated and times is the number of replications:

```r
rep(10,3)
```

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

```r
rep(c(0,5), 4)
```

```
## [1] 0 5 0 5 0 5 0 5
```

rep

```r
rep(1:5,2)
```

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

```r
rep(c('wx','yz'),3)
```

```
## [1] "wx" "yz" "wx" "yz" "wx" "yz"
```

If we want each component of a vector to be repeated before the next component, we add the option `each =` before the number of repetitions

```r
rep(c('wx','yz'), each = 3)
```

```
## [1] "wx" "wx" "wx" "yz" "yz" "yz"
```

rep

The number of replications can be a vector, in which case it must have the same number of components as the pattern, and then each element of the pattern is repeated the corresponding number of times.

```
rep(c(10,20),c(2,4))
```

```
## [1] 10 10 20 20 20 20
```

```
rep(1:3,1:3)
```

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

```
rep(1:3,rep(4,3))
```

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

rep

It is also possible to specify the number of terms the replications should have using the option `length`:

```r
rep(c(1,2,3,4), length=10)
```

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

Indexing

Indexing

To get a particular component from a vector, you can write the position of the element in the vector within square brackets after the vector's name, as in the following example:

```r
z <- 1:10
z[3]
```

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

Indexing

The general syntax for indexing is

```
object[index]
```

where `object` can be any data object in `R` (data objects will be reviewed later) and `index` can be any of the following alternatives:

1.- **Positive integers** corresponding to the position of the data points we are interested in.

2.- **Negative integers** that correspond to entries to be excluded:

3.- **Logical variables**. `TRUE` values will be included while `FALSE` values will not.

Indexing

1.- **Positive integers** corresponding to the position of the data points we are interested in.

For the following examples we use the data set `letter`, which has the 26 letters used in English.

```
letters[19]
```

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

```
letters[c(11,13,15)]
```

```
## [1] "k" "m" "o"
```

```
letters[seq(11,15,2)]
```

```
## [1] "k" "m" "o"
```

Indexing

2.- **Negative integers** that correspond to entries to be excluded:

```
letters[-(11:26)]
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
```

```
letters[-seq(1,25,2)]
```

```
##  [1] "b" "d" "f" "h" "j" "l" "n" "p" "r" "t" "v"
## [12] "x" "z"
```

Indexing

3.- **Logical variables**. `TRUE` values will be included while `FALSE` values will not.

```
a <- 1:26
a < 11
```

```
##  [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
##  [8]  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE
## [15] FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [22] FALSE FALSE FALSE FALSE FALSE
```

```
letters[a<11]
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
```

Indexing

```
a %% 2 == 0
```

```
##  [1] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE
##  [8]  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
## [15] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE
## [22]  TRUE FALSE  TRUE FALSE  TRUE
```

```
letters[a %% 2==0]
```

```
##  [1] "b" "d" "f" "h" "j" "l" "n" "p" "r" "t" "v"
## [12] "x" "z"
```

Indexing

Double inequalities such as $1 < x \leq 7$ should be written using the & symbol: `1 < a & a <= 7`:

```
letters[1 < a & a <= 7]
```

```
## [1] "b" "c" "d" "e" "f" "g"
```

Indexing

Here is a list of the comparison operators in R:

Symbol	Relation
$<$	less than
$>$	greater than
$<=$	less than or equal to
$>=$	greater than or equal to
$==$	equal to
$! =$	not equal to

Table 1. Comparison Operators.

Indexing

If the indexing vector is shorter than the object to which it is being applied, it is recycled:

```
letters[c(T,F)]
```

```
##  [1] "a" "c" "e" "g" "i" "k" "m" "o" "q" "s" "u"
## [12] "w" "y"
```

This gives the letters that occupy an odd position in the vector.

which

The function `which` gives a list of the positions within an object occupied by entries that satisfy a certain condition.

For the next example we use the data set `trees` which provides measurements of the `Girth`, `Height` and `Volume` of timber in 31 felled black cherry trees.

Girth is the diameter of the tree (in inches) measured at 4 ft 6 in above the ground.

The expression `tree$Volume` indicates the variable `Volume` of this data set.

```
str(trees)
```

```
## 'data.frame':    31 obs. of  3 variables:
##  $ Girth : num  8.3 8.6 8.8 10.5 10.7 10.8 11 11 11.1 11.2 ...
##  $ Height: num  70 65 63 72 81 83 66 75 80 75 ...
##  $ Volume: num  10.3 10.3 10.2 16.4 18.8 19.7 15.6 18.2 22.6 19.9 ..
```

which

```
which(trees$Volume > 50)
```

```
## [1] 26 27 28 29 30 31
```

```
trees[which(trees$Volume > 50),]
```

```
##    Girth Height Volume
## 26  17.3     81   55.4
## 27  17.5     82   55.7
## 28  17.9     80   58.3
## 29  18.0     80   51.5
## 30  18.0     80   51.0
## 31  20.6     87   77.0
```

```
trees[trees$Volume > 50,]
```

```
##    Girth Height Volume
## 26  17.3     81   55.4
## 27  17.5     82   55.7
## 28  17.9     80   58.3
## 29  18.0     80   51.5
## 30  18.0     80   51.0
## 31  20.6     87   77.0
```

Sampling

Sampling

The `sample` function generates random samples from a given set. The syntax is

```
sample(x, size, replace = FALSE, prob = NULL)
```

where

- `x` is the set (vector) from which we want to obtain the sample,
- `size` is the size of the sample,
- `replace` indicates if repeated values are allowed or not, and
- `prob` is a probability vector if we want to get a sample with a non-uniform distribution.

Sampling

```r
xy <- c('bad','regular','good')
sample(xy,10,replace=T)
```

```
##  [1] "bad"     "bad"     "regular" "bad"
##  [5] "regular" "regular" "regular" "bad"
##  [9] "regular" "bad"
```

```r
pp <- c(0.1,0.1,0.8)
sample(xy,10,replace=T,prob=pp)
```

```
##  [1] "bad"  "bad"  "good" "bad"  "good" "good"
##  [7] "good" "good" "good" "good"
```

Sampling

`R` also has functions for getting samples from many probability distributions.

The usual syntax of these functions is `r`*dist*, where *dist* designates the distribution; for example, to generate values from the normal distribution we use `rnorm`.

Depending on the distribution, it may be necessary to specify one or more parameters.

Table 2 in the next slide presents the most common distributions, the required parameters and their default values. `n` always represents the sample size.

Sampling

Distribution	**Function**
Gaussian	`rnorm(n, mean=0, sd=1)`
Exponential	`rexp(n, rate=1)`
Gamma	`rgamma(n, shape, scale=1)`
Poisson	`rpois(n, lambda)`
Weibull	`rweibull(n, shape, scale=1)`
Cauchy	`rcauchy(n, location=0, scale=1)`
Beta	`rbeta(n, shape1, shape2)`
t	`rt(n, df)`
Fisher	`rf(n, df1, df2)`
χ^2	`rchisq(n, df)`
Binomial	`rbinom(n, size, prob)`
Multinomial	`rmultinom(n, size, prob)`
Geometric	`rgeom(n, prob)`
Hipergeometric	`rhyper(nn, m, n, k)`
Logistic	`rlogis(n, location=0, scale=1)`
Lognormal	`rlnorm(n, meanlog=0, sdlog=1)`
Negative Binomial	`rnbinom(n, size, prob)`
Uniform	`runif(,n min=0, max=1)`

Table 2. Functions for generating random numbers.

Sampling

Let‘s look at some examples:

```
(rnorm(10))
```

```
##  [1] -1.5116091  1.3751693 -0.4357804 -1.7283232
##  [5] -0.6642065 -1.4781491 -0.5056712 -0.1952119
##  [9]  1.0769536 -1.3936343
```

```
(rbinom(5, 20, 0.5))
```

```
## [1]  6 12 12 12  9
```

```
(rexp(8))
```

```
## [1] 0.03823834 1.84348828 0.60447671 5.48553299
## [5] 0.41306232 0.59943733 0.29676255 0.18739561
```

```
(rpois(4, lambda=10))
```

```
## [1] 11  7  6 11
```

Distributions

Associated with these functions for generating random numbers are three others (for each distribution) that give values for the density, distribution and quantile functions.

The initial letter `r` must be replaced by `d, p` or `q`, respectively. The main argument for these functions is the vector of points where we want the function to be evaluated.

```
pnorm(1.96)
```

```
## [1] 0.9750021
```

```
qnorm(0.975)
```

```
## [1] 1.959964
```

Distributions

The quantiles for the normal distribution for a two-sided test at the 5% level are

```
qnorm(c(0.025,0.975))
```

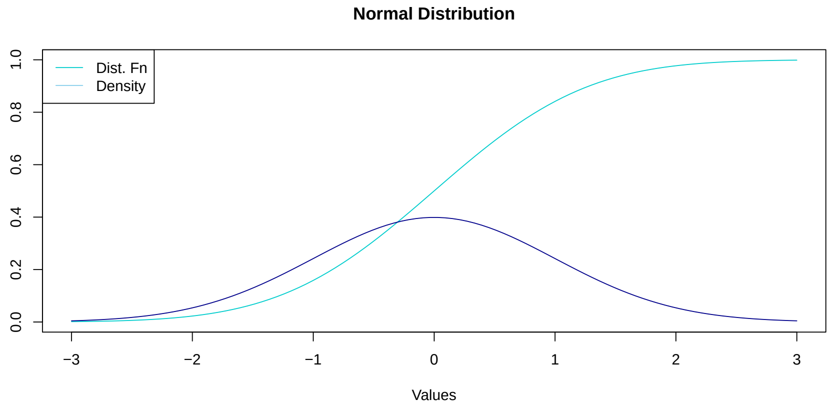
```
## [1] -1.959964  1.959964
```

Graph for the density and distribution function, standard normal distribution:

```
points.x <- seq(-3,3,length=100)
points.den <- dnorm(points.x)
points.fd <- pnorm(points.x)
```

Distributions

```r
plot(points.x, points.fd,type='l',xlab='Values',
     ylab='', main='Normal Distribution',col='cyan3')
lines(points.x,points.den,col='darkblue')
legend('topleft',c('Dist. Fn','Density'),
     col=c('cyan3','skyblue'),lty=rep(1,2))
```



Vectorized Operations

Vectorized Operations

R performs vector operations componentwise: if we add two vectors of equal length, the result is another vector of the same length, whose components are the sum of the components of the vectors we add. This is also true for any other arithmetic operation, including powers.

```
(a <- 5:2)
```

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

```
(b <- (1:4)*2)
```

```
## [1] 2 4 6 8
```

Vectorized Operations

```r
a - b
```

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

```r
a * b
```

```
## [1] 10 16 18 16
```

```r
a / b
```

```
## [1] 2.50 1.00 0.50 0.25
```

```r
a^b
```

```
## [1]  25 256 729 256
```

Vectorized Operations

Function evaluation is also a vectorized operation.

If, for instance, we want to evaluate the logarithm of the components of a certain vector `a`, we only have to write `log(a)`:

```r
log(a)
```

```
## [1] 1.6094379 1.3862944 1.0986123 0.6931472
```

Vectorized Operations

The following table lists functions commonly used with vectors.

Name	**Function**
`length(x)`	length of `x`
`sum(x)`	sum of the components of `x`
`prod(x)`	product of the components of `x`
`cumsum(x), cumprod(x)`	cumulative sum and product for `x`
`max(x), min(x)`	maximum and minimum of the components of `x`
`cummax(x), cummin(x)`	cumulative maximum and minimum of `x`
`sort(x)`	orders vector components
`rev(x)`	reverses the order of the elements of `x`
`diff(x)`	calculates the difference between successive components
`which(condition)`	gives the indices of the components that satisfy ‘condition‘
`which.max(x)`	index of the largest element
`which.min(x)`	index of the smallest element
`range(x)`	range of values for `x`
`mean(x)`	average of the elements of `x`
`median(x)`	median for the elements of `x`
`round(x,n)`	rounds the elements of `x` up to `n` decimals
`rank(x)`	ranks of the elements of `x`
`unique(x)`	vector with duplicate elements of `x` removed

Table 3. Vector Functions.

NAs and Infs

NAs and Infs

Missing data in R are denoted by NA (not available). When we do an operation with an NA the result will be an NA.

```
(y <- c(1:5,NA))
```

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

```
2*y
```

```
## [1]  2  4  6  8 10 NA
```

```
max(y)
```

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

Care must be taken with this but R has methods for dealing with missing data that will appear in the course as needed.

NAs and Infs

R can handle adequately the values $\pm\infty$ with $\pm$Inf or non-numerical values with NaN (not a number):

```
(x <- 2/0)
```

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

```
exp(x)
```

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

```
exp(-x)
```

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

```
x - x
```

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