My R notebook

Tandara Florin

8/30/2020

# Entering Input

At the R prompt we type expressions. The **<-** symbol of the assignment operator.

x <- 1  
print(x)

## [1] 1

x

## [1] 1

msg <- "hello"  
print(msg)

## [1] "hello"

The grammar of the language determines wether an expression is complete or not.

## x <- ## Incomplete expression

The **#** character indicates a comment. Anything to the right of the **#** (including **#** itself) is ignored.

# Evaluation

When a complete expression is entered at the prompt, it is evaluated and the result of the evaluate expression is returned. The result may be auto-printed.

x <- 5 ## nothing printed  
x ## auto- printing occurs

## [1] 5

print(x) ## explicit printing

## [1] 5

The [1] indicates that x is a vector and 5 is the first element.

# Printing

x <- 1:20  
x

## [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

The **:** operator is used to create integer sequences.

# Objects

R has five basic or “atomic” classes of objects:

* character
* numeric (real numbers)
* integer
* complex
* logical (True/False)

The most basic object is a vector

* A vector can only contain objects of the same class
* BUT: The one exception is a *list*, which is represented as a vector but can only contai nobjects of different classes (indeed, that’s usually why we use them)

Empty vectors can be created with the **vector( )** function.

vector(mode = "integer", length = 0L)

## integer(0)

# Numbers

* Numbers in R are generally treated as numeric objects (i.e. double precision real numbers)
* If you explicitly want an integer, you need to sepcify the **L** suffix
* EX: Entering **1** gives you a numeric object; entering **1L** explicitly gives you an integer.
* there is also a special number **Inf** which represents infinity; e.g. **1 / 0**; **Inf** can be used in ordinary calcualtions; e.g. **1 / Inf** is **0**
* The value NaN represents an undefined value (“not a number”); e.g. **0 / 0**; **NaN** can also be tought of as a missing value (more on that later)

1

## [1] 1

1L

## [1] 1

n <- 1 / 0  
print(n)

## [1] Inf

m <- 1 / Inf  
print(m)

## [1] 0

c <- 0 / 0  
print(c)

## [1] NaN

# Attributes

R objects can have attributes:

* names, dimnames
* dimensions (e.g. matrices, arrays)
* class
* length
* other user-defined attributes/metadata

Attributes of an object can be accessed using the **attributes( )** function.

# Creating Vectors

The c( ) function can be used to create vectors

x <- c(0.5, 0.6) ##numeric  
x <- c(TRUE, FALSE) ##logical  
x <- c(T, F) ##logical  
x <- c("a", "b", "c") ##character  
x <- 9:29 ##integer  
x <- c(1+0i, 2+4i) ##complex

Using the **vector( )** function

x <- vector("numeric", length = 10)  
x

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

# Mixing Objects

What about the following?

y <- c(1.7, "a") ##character  
y <- c(TRUE, 2) ##numeric  
y <- c("a", TRUE) ##character

When different objects are mixed in a vector, *coercion* occurs so that every element in the vector is of the same class.

# Explicit Coercion

Objects can be explicitly coerced from one class to another using **as.\*** functions , if available.

x <- 0:6  
class(x)

## [1] "integer"

as.numeric(x)

## [1] 0 1 2 3 4 5 6

as.logical(x)

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

as.character(x)

## [1] "0" "1" "2" "3" "4" "5" "6"

Nonsensical coercion results in **NAs**.

x <- c("a", "b", "C")  
as.numeric(x)

## Warning: NAs introduced by coercion

## [1] NA NA NA

as.logical(x)

## [1] NA NA NA

as.complex(x)

## Warning: NAs introduced by coercion

## [1] NA NA NA

Now I mentioned lists a little bit earlier in this lecture and the idea is that they’re, they’re like a vector except that every element of a list could be a, an object of a different class and so that makes lists very, very handy for kind of carrying around different types of data. And they’re very useful in R and they become very common especially when in conjunction with other types of functions that we’re going to learn about.

x <- list(1, "a", TRUE, 1 + 4i)  
x

## [[1]]  
## [1] 1  
##   
## [[2]]  
## [1] "a"  
##   
## [[3]]  
## [1] TRUE  
##   
## [[4]]  
## [1] 1+4i

So lists are indexed you’ll notice that elements of a list will have double brackets around them elements of other vectors just have the single brackets, so that’s one way to separate a list from other types of vectors.

# Matrices

Matrices are vectors with a *dimension* attribute. The dimension attribute is itself an integer vector of length 2 (nrow, ncol)

m <- matrix(nrow = 2, ncol = 3)  
m

## [,1] [,2] [,3]  
## [1,] NA NA NA  
## [2,] NA NA NA

dim(m)

## [1] 2 3

attributes(m)

## $dim  
## [1] 2 3

# Matrices (cont’d)

Matrices are constructed column-wise, so entries can be tought of strting in the “upper left” corner and running down the columns.

m <- matrix(1:6, nrow = 2, ncol = 3)  
m

## [,1] [,2] [,3]  
## [1,] 1 3 5  
## [2,] 2 4 6

Matrices can also be created directly from vectors by adding a dimension attribute.

m <- 1:10  
m

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

dim(m) <- c(2, 5)  
m

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

# cbind-ing and rbind-ing

Matrices can be created by *column-binding or row-binding* with **cbind( )** and **rbind( )**.

x <- 1:3  
y <- 10:12  
cbind(x, y)

## x y  
## [1,] 1 10  
## [2,] 2 11  
## [3,] 3 12

rbind(x, y)

## [,1] [,2] [,3]  
## x 1 2 3  
## y 10 11 12

# Factors

Factors are used to represent data. Factors can be unordered or ordered. One can think of a factor as an integer vector where each integer has a *label*.

* Factors are treated specially by modelling functions **lm( )** and **glm( )**
* Using factors with labels is *better* than using integers because factors are self-describing; having a variable that has values “Male” and “Female” is better than variable that has values 1 and 2.

So one, you can think of a factor as an integer vector where each integer has a label. So for example, you might, you can think of it as a vector as one two three, where one represents you know, high, for example high value and two represents a medium value and three represents a low value. So you might have a, a variable that’s called high, medium and low. And underlying in R is represented by the numbers one, two, and three.

x <- factor(c("yes", "yes", "no", "yes", "no"))  
x

## [1] yes yes no yes no   
## Levels: no yes

table(x)

## x  
## no yes   
## 2 3

unclass(x)

## [1] 2 2 1 2 1  
## attr(,"levels")  
## [1] "no" "yes"

# Factors

The order of the levels can be be set using the *levels* argument to **factor( )**. This can be important in linear modelling because the first level is used as the baseline.

x <- factor(c("yes", "yes", "no", "yes", "no"), levels = c("yes", "no"))  
x

## [1] yes yes no yes no   
## Levels: yes no

This is important because in modeling functions and when you include a factor variable this, this, sometimes it’s important to know what the baseline level is. And so the baseline level is just the first level in the factor, and the way this is determined by R is critical. It’s determined using alphabetical order, so for example, if I create a factor variable with the elements yes and no, then the base line level will be the first level that’s encountered and because no comes before yes in the alphabet then **no will be the base line level** and **yes will be the second level**. Now this may not be something that you want. You might want for example a yes to be the base line level and no to be the second level and then in that case you have explicitly tell R that yes is going to be the first level and you can view that, using the levels argument to the factor function. So now when I print out the x object you see that the elements are still the same, still yes yes no, yes no. But the levels attribute is reversed. because yes is the first level and no is the second level.

# Missing Values

Missing values are denoted by NA or NaN for undefined mathematical operations.

* **is.na( )** is used to test objects if they are **NA**
* **is.nan( )** is used to test for **NaN**
* **NA** values have a class also, so there are integer **NA**, character NA, etc.
* A **NaN** value is also **NA** but the converse is not true

x <- c(1, 2, NA, 10, 3)  
is.na(x)

## [1] FALSE FALSE TRUE FALSE FALSE

is.nan(x)

## [1] FALSE FALSE FALSE FALSE FALSE

x <- c(1, 2, NaN, NA, 4)  
is.na(x)

## [1] FALSE FALSE TRUE TRUE FALSE

is.nan(x)

## [1] FALSE FALSE TRUE FALSE FALSE

# Data Frames

Data frames are used to store tabular data

* They are represented as a special type of list where every element of the list has to have the same length
* Each element of the list can be tought of as a column and the length of element of the list is the number of rows
* Unlike matrices, data frames can stored different classes of objects in each column (just like lists); matrices must have every element by the same class
* Data frames also have a special attribute called **row.names**
* Data frames are usually created by calling **read.table( )** or **read.csv( )**
* Can be converted to a matrix by calling **data.matrix( )**

x <- data.frame(foo = 1:4, bar = c(T, T, F, F))  
x

## foo bar  
## 1 1 TRUE  
## 2 2 TRUE  
## 3 3 FALSE  
## 4 4 FALSE

nrow(x)

## [1] 4

ncol(x)

## [1] 2

# Names

R objects can also have names, which is very useful for writing readable code and self-describing objects.

x <- 1:3  
names(x)

## NULL

names(x) <- c("foo", "bar", "norf")  
x

## foo bar norf   
## 1 2 3

names(x)

## [1] "foo" "bar" "norf"

Lists can also have names

x <- list(a = 1, b = 2, c = 3)  
x

## $a  
## [1] 1  
##   
## $b  
## [1] 2  
##   
## $c  
## [1] 3

And matrices. These are called **dimnames**.

m <- matrix(1:4, nrow = 2, ncol = 2)  
dimnames(m) <- list(c("a", "b"), c("foo", "bar"))  
m

## foo bar  
## a 1 3  
## b 2 4

A matrix from the sequence 1 to 4. It’s a two by two matrix. And so when I use the **dimnames( )** function I assign it a list. Where the first element of the list is the vector of row names and the second element of the list is a vector of column names.

#Summary

Data Types

* atomic classes: numeric, logical, character, integer, complex
* vectors, lists
* factors
* missing values
* data frames
* names