At the R prompt we type expressions.

The symbol (gets arrow/=) is the assignment operator.

>x 1

>print(x)

[1] 1

>x

[1] 1

>msg "hello"

======================================================

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

>x # Incomplete expression

The # character indicates a comment. Anything to the right of the # (including the # itself) is ignored. This is the only comment character in R. Unlike some other languages, R does not support multi-line comments or comment blocks.

=================================================

When a complete expression is entered at the prompt, it is evaluated and the result of the evaluated 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] shown in the output indicates that x is a vector and 5 is its first element.

Typically with interactive work, we do not explicitly print objects with the print function; it is much easier to just auto-print them by typing the name of the object and hitting return/enter.

However, when writing scripts, functions, or longer programs, there is sometimes a need to explicitly print objects because auto-printing does not work in those settings.

When an R vector is printed you will notice that an index for the vector is printed in square brackets []on the side. For example, see this integer sequence of length 20.

>x11:30

x

[1] 11 12 13 14 15 16 17 18 19 20 21 22

[13] 23 24 25 26 27 28 29 30

The numbers in the square brackets are not part of the vector itself, they are merely part of the printed output.

With R, it’s important that one understand that there is a difference between the actual R object and the manner in which that R object is printed to the console. Often, the printed output may have additional bells and whistles to make the output more friendly to the users. However, these bells and whistles are not inherently part of the object.

Note that 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 type of R object is a vector. Empty vectors can be created with the vector() function.

There is really only one rule about vectors in R, which is: **A vector can only contain objects of the same class**.

But of course, like any good rule, there is an exception, which is a *list*, which we will get to a bit later. A list is represented as a vector but can contain objects of different classes. Indeed, that’s usually why we use them.

There is also a class for “raw” objects, but they are not commonly used directly in data analysis and we won’t cover them here.

Numbers in R are generally treated as numeric objects (i.e. double precision real numbers). This means that even if you see a number like “1” or “2” in R, which you might think of as integers, they are likely represented behind the scenes as numeric objects (so something like “1.00” or “2.00”). This isn’t important most of the time…except when it is.

If you explicitly want an integer, you need to specify the L suffix. So entering 1 in R gives you a numeric object; entering 1L explicitly gives you an integer object.

There is also a special number Inf which represents infinity. This allows us to represent entities like 1 / 0. This way, Inf can be used in ordinary calculations; e.g. 1 / Inf is 0.

The value NaN represents an undefined value (“not a number”); e.g. 0 / 0; NaN can also be thought of as a missing value (more on that later)

Note that in the above example, T and F are short-hand ways to specify TRUE and FALSE. However, in general one should try to use the explicit TRUE and FALSE values when indicating logical values. The T and F values are primarily there for when you’re feeling lazy.

You can also use the vector() function to initialize vectors.

The c() function can be used to create vectors of objects by concatenating things together.

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

Note that in the above example, T and F are short-hand ways to specify TRUE and FALSE. However, in general one should try to use the explicit TRUE and FALSE values when indicating logical values. The T and F values are primarily there for when you’re feeling lazy.

You can also use the vector() function to initialize vectors.

>x vector("numeric", length = 10)

x

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

There are occasions when different classes of R objects get mixed together. Sometimes this happens by accident but it can also happen on purpose. So what happens with the following code?

y c(1.7, "a") ## character

y c(TRUE, 2) ## numeric

y c("a", TRUE) ## character

In each case above, we are mixing objects of two different classes in a vector. But remember that the only rule about vectors says this is not allowed. When different objects are mixed in a vector, coercion occurs so that every element in the vector is of the same class.

In the example above, we see the effect of implicit coercion. What R tries to do is find a way to represent all of the objects in the vector in a reasonable fashion. Sometimes this does exactly what you want and…sometimes not. For example, combining a numeric object with a character object will create a character vector, because numbers can usually be easily represented as strings.

Objects can be explicitly coerced from one class to another using the 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"

Sometimes, R can’t figure out how to coerce an object and this can result in NAs being produced.

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

Matrices are vectors with a dimension attribute. The dimension attribute is itself an integer vector of length 2 (number of rows, number of columns)

>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 are constructed column-wise, so entries can be thought of starting 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

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

>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

Lists are a special type of vector that can contain elements of different classes. Lists are a very important data type in R and you should get to know them well. Lists, in combination with the various “apply” functions discussed later, make for a powerful combination.

Lists can be explicitly created using the list() function, which takes an arbitrary number of arguments.

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

>x

[[1]]

[1] 1

[[2]]

[1] "a"

[[3]]

[1] TRUE

[[4]]

[1] 1+4i

We can also create an empty list of a pre specified length with the vector() function.

x vector("list", length = 5)

>x

[[1]]

NULL

[[2]]

NULL

[[3]]

NULL

[[4]]

NULL

[[5]]

NULL

Factors are used to represent categorical data and can be unordered or ordered. One can think of a factor as an integer vector where each integer has a label. Factors are important in statistical modelling and are treated specially by modelling functions like 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 a variable that has values 1 and 2.

Factor objects can be created with the factor() function.

>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

## See the underlying representation of factor

unclass(x)

[1] 2 2 1 2 1

attr(,"levels")

[1] "no" "yes"

Often factors will be automatically created for you when you read a dataset in using a function like read.table(). Those functions often, as a default, create factors when they encounter data that look like characters or strings.

The order of the levels of a factor can be set using the levels argument to factor(). This can be important in linear modelling because the first level is used as the baseline level. This feature can also be used to customize order in plots that include factors, since by default factors are plotted in the order of their levels.

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

x ## Levels are put in alphabetical order

[1] yes yes no yes no

Levels: no yes

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

levels = c("yes", "no"))

x

[1] yes yes no yes no

Levels: yes no

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

## Create a vector with NAs in it

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

## Return a logical vector indicating which elements are NA

is.na(x)

[1] FALSE FALSE TRUE FALSE FALSE

## Return a logical vector indicating which elements are NaN

is.nan(x)

[1] FALSE FALSE FALSE FALSE FALSE

## Now create a vector with both NA and NaN values

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 are used to store tabular data in R. They are an important type of object in R and are used in a variety of statistical modeling applications. Hadley Wickham’s package [dplyr](https://github.com/hadley/dplyr" \t "_blank) has an optimized set of functions designed to work efficiently with data frames, and ggplot2 plotting functions work best with data stored in data frames.

Data frames 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 thought of as a column and the length of each element of the list is the number of rows.

Unlike matrices, data frames can store different classes of objects in each column. Matrices must have every element be the same class (e.g. all integers or all numeric).

In addition to column names, indicating the names of the variables or predictors, data frames have a special attribute called row.names which indicate information about each row of the data frame.

Data frames are usually created by reading in a dataset using the read.table() or read.csv(). However, data frames can also be created explicitly with the data.frame() function or they can be coerced from other types of objects like lists.

Data frames can be converted to a matrix by calling data.matrix(). While it might seem that the as.matrix() function should be used to coerce a data frame to a matrix, almost always, what you want is the result of 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