The R building blocks

January 15, 2019

The interpreter

Let's start using the interpreter to get used to the basic arithmetic operations. They all work the way you expect to.

```
1 + 1
sqrt(2)
exp(1)
2^3
```

The same happens with comparisons:

```
1 > 1
0 == log(1)
(1 + 1 == 2) | (1 + 1 == 3)
(1 + 1 == 2) & (1 + 1 == 3)
```

We can assign values to a name using the *assignment operator* `<-`. R also supports `=` for assignment, but it is not recommended. For instance, we can now assign the value `1` to the variable `a`, and every time we call `a` we will get the value it is referred to:

```
a <- 1
a
```

Data types

R offers a variety of data types. The most relevant ones are: *numeric*, *integer*, *character*, and *logical*.

```
1.0 # Numeric
1L # "Integer"
"ab" # Character
TRUE # logical
```

The `#` indicates a comment, which may or may not be at the beginning of the line.

It also offers logical constants for missing value (`NA`), "Not a Number" (`NaN`) and positive and negative infinity (`Inf` and `-Inf`):

```
NA
0/0
1/0
```

You can test the type of an object using the `is.` functions (*"is `x` an integer?"*):

```
is.numeric(1.0)
is.integer(1) ## Integer is actually 1L
is.character("ab")
is.logical(FALSE)
```

A common mistake that we all make at some point (but hopefully you will not!) is to test whether an object is `NA` by using the `==` comparison.

```
NA == NA
```

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

There are good reasons for these "odd" behavior. Think of NA as meaning "I don't know what's there". The correct answer to 3 > NA is obviously NA because we don't know if the missing value is larger than 3 or not. Well, it's the same for NA == NA. They are both missing values but the true values could be quite different, so the correct answer is "I don't know."

We have special functions to check whether an object is `NA`:

```
is.na(NA)
```

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

and the same applies for `NaN`.

Note that `is.numeric` is a *function,* and all functions work the same way in R. They have a name followed by the argument(s) enclosed in parenthesis: `any_function(x)`. We will see a lot more about them in the following sessions.

Data structures

The most common (and intuitive) data structures are `vector`, `matrix`, `data.frame` and `list`. We will see `data.frame` in the next script.

Atomic Vectors

Vectors are an ordered collection of objects, and atomic vectors are vectors which contain only one data type. Atomic vectors can be created with the function `c`, which stands for "concatenate".

```
c(1, 2)
c(1, 2, NA)
c(1, "a") # See how the first element is casted into character!
```

There are a few functions that allows us to create atomic vectors with particular structures. For instance, we can create a sequence using the function `seq`:

```
c(1, 2, 3, 4)
seq(1, 4)
1:4 # A shorthand for `seq`
```

See how we used a function with two (positional) arguments? It is probably a good moment to take a look at more information about `seq` using `?seq`

Lists

Unlike atomic vectors, lists may contain multiple data types and may even contain other data structures as elements.

```
list("first"=c(1, 2, 3), "second"=c("a", "b"), "third"= list("one" = 1, "a" = "a"))
```

Matrices & Arrays

A `matrix` is a rectangular table of elements of the same type.

```r
A <- matrix(c(1, 2, 3, 4), nrow=2, ncol=2)
A
```

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

Note how we passed all data as a unique object (a `vector`). Also notice how the matrix is filled. Contrary to intuition (but it makes perfect sense, if you think about it) is that a 1-dimensional `matrix` is not a `vector`.

Operations in matrices work "kind of" the way you would expect:

```r
B <- matrix(c(4, 3, 2, 1), nrow=2)
A * B
```

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

```r
A %*% B
```

```
##      [,1] [,2]
## [1,]   13    5
## [2,]   20    8
```

See how the first operation is element-wise, while the second is the standard matrix multiplication. We also have common operations like extracting the diagonal (which is a `vector`) or calculating the transpose or the inverse:

```r
t(A)
diag(A)
solve(A) # A^-1
```

An `array` is a generalization of matrices to any number of dimensions.

Determining the size of data structures

The "length" function is used to determine the length of vectors, and the "dim" function is used to determine the dimensions of a matrix, array, or data frame. Further, the 'nrow' and 'ncol' functions can be used to access the number of rows or columns in a matrix or data frame.

```r
x <- 1:5
length(x)
```

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

```r
x <- matrix(1:4, nrow = 2)
dim(x)
```

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

```r
nrow(x)
```

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

Everything is a vector

R has a certain kind of purity to it. In particular:

- Everything that happens is a function call.
- Everything that exists is an object.
- Everything that exists is also a vector!

Scalars do not exist in R. The number 5, for example, is a vector of length 1.

Higher-dimensional structures like matrices and arrays are merely vectors with a dimension attribute attached.

```
X <- 1:4
class(X)
```

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

```
dim(X) <- c(2,2)
class(X)
```

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

Naming elements of data structures

Optionally, elements of vectors can be named. In practice, elements of lists are usually named, and elements of atomic vectors are usually not, but this is a matter of convention and style.

The "names" function can be used to extract the names of the vector elements.

```
x <- list(first = 1:5, second = "hello")
names(x)
```

```
## [1] "first"  "second"
```

You can also change the names of vector elements as follows.

```
names(x) <- c("a", "b")
```

Indexing

Indexing allows us to access elements of a data structure by position:

```
a <- c(1, 2, 3)
a[1]
a[1:2]
a[-3]
a[1] <- 0
```

A matrix is two dimensional, so we need two elements to indicate cells in our structure but it works using the same principles:

```
A <- matrix(c(1, 2, 3, 4), nrow=2)
A[1, 1]
A[1, ] # Empty space is short for "everything in this dimension"
A[, 1]
A[2, 2] <- 0
A[2, ] <- 0 # Recycling rule
```

A list uses a similar notation, but with two brackets instead of one. There are other ways to access elements.

```
my_list <- list("a"=1, "b"=2, "c"=NA)
my_list[["a"]] # Name
```

```
my_list[[1]] # Position
my_list$a <- c("Not 1 anymore")
```

Notice how we can access elements by name or position. Elements can also be accessed by logical conditions.

```
x <- 1:5
x[x < 3]
```

The use of single vs. double brackets in R confuses new users. However, the rules are simple. Double brackets are used for accessing single elements from a data structure. Single brackets are for accessing any number of elements from the data structure, and the result is of the same structure (but possibly different length) as the original structure. For accessing individual elements of an atomic vector, there is no difference between single and double brackets, because the elements of an atomic vector are themselves atomic vectors (remember: there are no scalars in R), so single brackets are typically used in this case.

```
a <- list(1, 2)
class(a[1])
```

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

```
class(a[[1]])
```

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

```
b <- 1:2 # an atomic vector
class(b[1])
```

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

```
class(b[[1]])
```

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

Data frames

`data.frame` is the building block for most of what we will do in data analysis. Think about them as a `matrix` that can hold columns of different types and with column names. Data frames are much like SAS data sets.

```
states <- data.frame("code"       = c("CA", "NY", "NE", "AZ"),
                     "population" = c(38.8, 19.7, 2.1, 6.8),
                     "region"     = c("West", "Northeast", "Midwest", "West"),
                     "landlock"   = c(FALSE, FALSE, TRUE, TRUE))
```

We can access elements via indexing the same way as we would do in a `matrix` or we can access by name:

```
states[, 3]
states[, "region"]
states$region
```

We can also add new variables:

```
states$spanish <- c(28.5, 15.7, NA, 19.5)
```

Transformation of variables is straightforward:

```
states$spanish <- states$spanish * states$population # But we could have used a new variable
```

But we will see soon that we don't need to do most of this transformations.

The same approach can be used for subsetting a dataset, i.e., selecting rows from a `data.frame` based on given values:

```r
states[states$population > 10,] # Notice the comma!
subset(states, population > 10)
```

Let's take the code apart. We want to take the dataset `states` and keep only the rooms for which the statement `states$population > 10` is true. As we saw before, the comparison actually returns a logical vector that has value `TRUE` when the condition is met. We can then select rows from the data by simply passing this logical vector to the dimension that captures the rows.

We can use the same strategy to do recoding:

```r
states$population[states$code == "NE"] <- 1.8
```

Carefully examine what's happening in the previous line. `states$code` gets the column `code` in our dataset. `states$code == "NE"` returns a logical vector in which only the third observation will be `TRUE`. `states$population[states$code == "NE"]` access the `population` value for Nebraska. And then we assign the correct value 1.8 to it. The underlying idea is that we select the observations that we want to transform and replace them.

The dataset is also useful to think about variable types. Consider the variable `region`: it is a vector of characters, but we would like to consider it as a discrete variable in which we would represent it as a value (a number) with a label (the region name). That's a `factor` in R.

```r
states$region <- factor(states$region)
states$region
levels(states$region)
```

Depending on who you talk to, they may hate them or love them.

Conditionals

As you can imagine, conditionals execute an action depending on the value (`TRUE` or `FALSE`) of another statement. In R, they use the follow syntax:

```r
if (x > 0) {
  print("It's positive")
} else if (x == 0) {
  print("It's neither positive nor negative")
} else {
  print("It's negative")
}
```

Naming variables

Variable names in R can consist of any ASCII letter or number, as well as periods and underscores. Further, they may not begin with an underscore, be equal to any R keyword (e.g., if, function), or be over 10,000 characters long (I hope your variable names are not even close to that). By surrounding a variable name in backticks, most of these rules can be broken, but it is not recommended.

Variable names should be as descriptive as possible (unlike my examples above!). For variables which consist of multiple words, there is unfortunately no single convention. Three different styles are common:

- my_variable_name
- myVariableName
- my.variable.name

I personally use the first. Some prefer the second. The third is often see too, but I don't recommend it since periods have semantic meaning in R. However, the important part is to be consistent in whatever you choose.