Advanced R

Yu Chen

August 29, 2017

Data Structures

1. What are the six types of atomic vectors? How does a list differ from an atomic vector?

The six types of atomic vectors are **logical**, **integer**, **double** (or **numeric**), **character**, **complex**, and **raw**. All elements of an atomic vector must be the same type, whereas the elements of a list can be of different types.

2. What makes `is.vector()` and `is.numeric()` fundamentally different to `is.list()` and `is.character()`?

The function **`is.vector()`** encompasses **`is.list()`**, since a **list** object in R is a type of vector. If you attempt to create an **atomic vector** using different types, the object created will be coerced into **character** type:

```
vec <- c(1,2,3)
is.vector(vec) # TRUE
is.list(vec) # FALSE
is.vector(vec, mode = "list") # FALSE
is.character(vec) # FALSE
is.numeric(Vec) # TRUE
```

You can see here than an atomic vector of integers is a **vector**, not a **list**, and also not of the **character** type. It is, however, **numeric**.

```
vec2 <- list(1,2,3)
is.vector(vec2) # TRUE
is.list(vec2) # TRUE
is.character(vec2) # FALSE

vec3 <- c(1,"ab", TRUE)
is.vector(vec3) # TRUE
is.list(vec3) # FALSE
is.character(vec3) # TRUE
str(vec3)
```

The function `is.vector()` does not actually test if an object is a vector. It returns `TRUE` only wif the object is a vector with no attributes apart from names. That is why you should use `is.atomic(x) || is.list(x)` to test if an object is actually a vector.

3. Test your knowledge of vector coercion rules by predicting the output of the following uses of `c()`:

```
c(1, FALSE)
c("A", 1)
c(list(1))
c(TRUE, 1L)
```

The first example, **`c(1, FALSE)`**, will return an atomic vector with two elements: `1`, and `0`, since `FALSE` can be coerced into an integer.

The second example, **`c("A", 1)`**, will be coerced into characters, since it is of both `character` and `integer` type.

The third example, **`c(list(1))`** will return an atomic vector with one element, a `list`.

The fourth example, **`c(TRUE, 1L)`** will return an atomic vector with two elements, both of `1`. As a side note, what does the `L` after a number mean? It means that the number is an **explicit integer**. There are some use cases for this- for example, with performance, since a number stored as a `double` takes up 8 bytes of memory, whereas explicitly storing a number as an integer uses only 4 bytes. A great explanation can be found in this StackOverflow answer:

```
x <- 1:100
typeof(x) # integer

y <- x+1
typeof(y) # double, twice the memory size
object.size(y) # 840 bytes (on win64)

z <- x+1L
typeof(z) # still integer
object.size(z) # 440 bytes (on win64)
```

4. Why do you need to use `unlist()` to convert a list to an atomic vector? Why does `as.vector()` not work?

Compare the two outputs when attempting to convert a list into an atomic vector:

```
vec2 <- list(1,2,3)
unlist(vec2)

## [1] 1 2 3

as.vector(vec2)

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

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

As you can see, `as.vector` has not actually "flattened" the list. It's made a vector of three mini-lists. As a side note, the notation `[[1]]` can be confusing. This StackOverflow answer explains that `[[ ]]`

```
extracts a single element by name or position from a list or data frame. For
example, iris[["Sepal.Length"]] extracts the column Sepal.Length from the
data frame iris; iris[[2]] extracts the second element from iris.
```

Thus, the element itself it is still a list:

```
typeof(as.vector(vec2)[1])
```

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

Moreover, `unlist()` wil attempt to preserve naming information as much as possible, as this example shows:

```
test1 <- list(5, "b", 12)
unlist(test1)
```

```
## [1] "5"  "b"  "12"
```

```
test2 <- list(v1=5, v2=list(2983, 1890), v3=c(3, 119))

unlist(test2)
```

```
##   v1  v21  v22  v31  v32
##    5 2983 1890    3  119
```

5. Why is 1 == "1" true? Why is -1 < FALSE true? Why is "one" < 2 false?