R Names and Lists

We'll be covering names in this lesson and moving into lists & dataframes. This will give greater insight into how to access the internals of various data structures in R.

Names

Elements in atomic vectors can have "names" associated to them. The most intuitive way to demonstrate how names work is to simply illustrate how they're assigned. The two most commonly used ways of assigning names are:

- Creating a vector with names directly: `c(name1 = 1, name2 = 2 name3 = 3)`
- Calling `names()` on an existing vector and modifying it: 'x = 1:3; names(x) = c("name1", "name2", "name3")

Try it out in the console – define a vector, assign names to its values, and then print out the new vector to see what shows up.

Exercise. Can multiple values be named the same thing?

Exercise. What do you get if you try to access the names of an unnamed vector?

Exercise. Are there any type restrictions on what names can be? What happens if you assign a logical vector of names and print out the names afterward?

Exercise. What happens when the vector to which you assign to `names()` is shorter than the underlying vector? Longer?

You can remove the names associated with a vector `x` by using `unname(x)` or setting `names(x) = NULL`.

Names provide a convenient way of accessing the values of a vector. We'll cover vector subsetting in greater detail later, but for now know that you can do the following:

```
> x = c(a = 1, b = 2, c = 3)
> x["a"]
a
1
> x[c("c", "b")]
c b
3 2
```

Play around with the above and make sure you understand how it works.

Exercise. What happens when more than one element has the same name?

Lists

Similar to atomic vectors, lists are another data structure in R. The main differences are:

- Lists can be nested within each other.
- Lists can contain many different data types, not just a single data type.

For instance, we may make a new list with `x = list("a", 1, TRUE)`. An empty list is constructed with just `list()`.

Exercise. What is the data type of a list?

To turn a list into an atomic vector, you can use `unlist()`.

Exercise. Write a function `combine(a, b)` that takes lists `a` and `b`, returning a single list with both the elements of `a` and the elements of `b`. *E.g.*, `combine(list(1, 2), list(3, 4),` would return `list(1, 2, 3, 4).` `(*Hint:* Try using`c()'.)

Exercise. What happens when you `unlist()` a nested list?

List elements can also be named, just like with atomic vectors, and can be accessed similarly. There are, however, *nuances* to list subsetting that we'll cover in depth later.

Exercise. You can access a named element in a list with, *e.g.*, `x$a`. What's the difference between `x["a"]` and `x$a` (if you have, say, `x = c(a = 1, b = 2, c = 3)`)?[1]

Exercise. What happens when you try to combine vectors with lists? Lists inside vectors? Vectors inside lists?

Subsetting with lists

We'll cover this in greater depth in the next lesson. Briefly, when accessing the items of a list with, say, `L[1:5]`, what's returned isn't the underlying items – because there's no way to guarantee that they're the same type, returning a vector would be very likely to introduce unwanted coercion (especially considering that the use case of a list is when the contents have different types). Rather, a direct single-bracket subset of a list returns the subset of items contained in a smaller list.

Keep this in mind as you progress – we'll soon learn how to modify the contents of a list directly.

[1] Suppose that we have `x = list(a = 1, b = 2, c = 3)`. Then accessing `x["a"]` returns a **list** equivalent to `list(a = 1)`, whereas accessing `x$a` accesses the **value within**, equal to `1`. With atomic vectors, this doesn't make a difference, because a single value is *equivalent* to a vector of length 1, but this isn't the case with lists!

Data frames

You'll be constantly working with data frames in R; it's a convenient structure to store all kinds of data.

Here's the main thing to take away from this section: **data frames are built on top of lists!** Keep this in mind as you work with data frames. They're nothing more than a class built on top of lists, where each list element is a vector constrained to be the same length as the others in the data frame (along with some other bells and whistles). The behavior of data frames can seem opaque and confusing at first, but it becomes less so as you understand how R's data structures work internally.

A data frame is *two-dimensional*, with both *rows* and *columns*, which changes things around. They can be created with `data.frame()`, *e.g.*, `df = data.frame(x = 1:3, y = c(TRUE, FALSE, TRUE))`.

In the following examples, it may be helpful to have a small but nontrivial data frame object to play around with, so you can set:

```
df = data.frame(matrix(1:100, nrow=10, ncol=10))
```

This will assign a dataframe to `df` with a simple structure (so how different operations work on the dataframe will be more apparent). For now, don't worry about how the `matrix` command works – matrices are a part of R, but they aren't really very important, so we'll cover them later.[2]

Exercise. What are the *type* and *class* of a data frame?

You'll notice that by default both the rows *and* the columns of a data frame have labels! You can access them with `rownames()` and `colnames()`, which work in the same way as `names()`.

Exercise. Does `names()` return the column or row names of a data frame?

Remark. The dimensions of a data frame can also be accessed with `nrow()` and `ncol()`.

Exercise. You can use either `data.frame()` or `as.data.frame()` to convert existing data to data frames (the differences between the two are trivial). Try converting vectors and lists into data frames. What behavior do you observe? What happens when the elements of a list are of different lengths?

Sometimes, you'll want to combine two data frames into the same one.

Exercise. Using the 10-by-10 data frame defined earlier, use `rbind()` and `cbind()` to make 10-by-20 and 40-by-10 data frames, verifying the dimensions with `dim()`. *Hint:* You can do this without nesting `rbind()` calls within `rbind()` calls or `cbind()` calls within `cbind()` calls.

[2] Lots of R tutorials introduce matrices early. It's confusing and, in practice, matrices aren't that useful for doing basic data analysis.

Exercise. The `do.call(func, args)` function is very useful – suppose that `args = c(1,2,3)`; then `do.call(func, args)` is equivalent to calling `func(1,2,3)`. Combining `do.call()` with `rep()` and our previously defined 10-by-10 data frame, write a very short line to create a 10-by-100 data frame.[3]

Remark. You can have a list as a column of a data frame, or even matrices and arrays, but these occurrences are *very* infrequent. Most functions that accept data frames as input will assume, without checking, that every column is an atomic vector.

Exercise. Play around with what happens when you pass in a vector of *characters* when creating a data frame.[4]

Supplementary Exercises

Advanced R, 2.1.3.4. Why do you need to use `unlist()` to convert a list to an atomic vector? Why doesn't `as.vector()` work?[5]

Advanced R, 2.4.5.3. Can you have a data frame with 0 rows? What about 0 columns?

[3]You may notice some disturbingly flexible instances of type coercion. This is pretty much an unavoidable part of programming for data science.

[4]By default, `data.frame()` coerces vectors of strings into *factors*. (Those will be covered later.) To disable this behavior, pass in the parameter `stringsAsFactors=FALSE`.

[5]Technically, lists *are* vectors… but this is just a technicality of the language—lists are vectors, but not *atomic* vectors (try `is.vector()` and `is.atomic()` on a list). You can't use `as.atomic()` because it doesn't exist.