R: Data frames

Signal Data Science

This lesson will cover the details of some more advanced 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:

1. Creating a vector with names directly: `c(name1 = 1, name2 = 2 name3 = 3)`
2. 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.

- Can multiple values be named the same thing?
- What do you get if you try to access the names of an unnamed vector?
- 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?
- 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.

- 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:

1. Lists can be nested within each other.
2. 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()`.

- What is the data type of a list?

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

- 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()`.)
- What happens when you `unlist()` a nested list?
- Why do you need to use `unlist()` to convert a list to an atomic vector? Why doesn't `as.vector()` work?[1]

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.

- 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)`)?[2]
- What happens when you try to combine vectors with lists? Lists inside vectors? Vectors inside lists?

[1] 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.

[2] 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!

Subsetting with lists

We'll cover this in greater depth in a future section. 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.

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))
```

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

- 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()`.

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

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

- 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.

- 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 me defined earlier, use `rbind()` calls within me defined earlier, use `rbind()` calls or `cbind()` calls within `cbind()` calls.
- 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]

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.

- Play around with what happens when you pass in a vector of *characters* when creating a data frame.[4]
- Can you have a data frame with 0 rows? What about 0 columns?

Subsetting

"Subsetting" refers to the act of getting a "subset" of a list, vector, or other structure in R.

Wickham writes:

> Subsetting is a natural complement to `str()`. `str()` shows you the structure of any object, and subsetting allows you to pull out the pieces that you're interested in.

Keep the `str()` function in mind as you work through this lesson.

[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`.

Simple single-bracket subsetting

The simplest form of subsetting uses the single brackets `[` and `]`. We'll cover how this works with a variety of data types. Sometimes, we'll refer to the vector used to subset a different vector as the *index vector*. (For example, in `x[values]`, `values` is the index vector.)

There are three main ways to subset an unnamed vector—figure out how they work by playing around with `x = 1:5`.

1. Subsetting with positive integers: `x[c(3,1)]`
2. Subsetting with negative integers: `x[-c(3,1)]`, `x[c(-3,-1)]`
3. Subsetting with logical vectors: `x[c(TRUE, FALSE, TRUE, FALSE, TRUE)]`

- What happens when you subset with multiple copies of the same positive integer?
- What happens when you subset with numbers that are not whole numbers?
- What happens when you subset with both positive and negative integers?
- What happens when you subset with a logical vector shorter than the vector you're subsetting? Try with short logical vectors of length 1 and 2.
- What happens when there are some `NA` values in the index vector?

Moreover, you can pass in *character vectors* as the index vector to subset based on *names*.

- What happens if you try to subset by name but one of the values you pass in isn't a valid name?
- Use `[]` to play around with subsetting lists. What *type* of object do you get back?
- Write code to replace the 3rd item in `list(1, 2, 3, 4, 5)` with `1:5`, resulting in `list(1, 2, 1:5, 4, 5)`. Use single-bracket subsetting.

A list `L` can be extended by simply assigning items to an index greater than `length(L)`.

- Make a list of length 10. What happens when you assign a value to index 20? In general, is it faster to initialize a list with arbitrary values and iteratively fill in those values with the right ones or to simply iteratively extend a preexisting empty list with the right values? Experiment with lists and the `tictoc` package to find out.

Recall that data frames are simply more complicated versions of lists of vectors. If you subset with a *single vector*, data frames behave identically to lists. However,

you can simultaneously subset both dimensions by passing in *two* vectors. It's easiest to demonstrate:

```
> df = data.frame(matrix(1:100, nrow=10, ncol=10))
> df
   X1 X2 X3 X4 X5 X6 X7 X8 X9 X10
1   1 11 21 31 41 51 61 71 81  91
2   2 12 22 32 42 52 62 72 82  92
3   3 13 23 33 43 53 63 73 83  93
4   4 14 24 34 44 54 64 74 84  94
5   5 15 25 35 45 55 65 75 85  95
6   6 16 26 36 46 56 66 76 86  96
7   7 17 27 37 47 57 67 77 87  97
8   8 18 28 38 48 58 68 78 88  98
9   9 19 29 39 49 59 69 79 89  99
10 10 20 30 40 50 60 70 80 90 100
> df[2:4, 3:6]
  X3 X4 X5 X6
2 22 32 42 52
3 23 33 43 53
4 24 34 44 54
```

Now, some exercises:

- Does subsetting a data frame with a single vector select rows or columns?
- What happens when you pass in nothing for one of the two vectors, like with `df[1:2,]` or `df[,5:6]`?
- When subsetting with two vectors, can you pass in a vector of column names?
- What's the difference between (1) subsetting a single column by passing in a single number as the index vector versus (2) subsetting a single column by passing in nothing for the first index vector and a single number for the second index vector?

Advanced subsetting

Wickham writes:

> There are two other subsetting operators: `[[` and `$`. `[[` is similar to `[`, except it can only return a single value and it allows you to pull pieces out of a list. `$` is a useful shorthand for `[[` combined with character subsetting.
>
> You need `[[` when working with lists. This is because when `[` is applied to a list it always returns a list: it never gives you the contents of the list. To get the contents, you need `[[`:

> "If list `x` is a train carrying objects, then `x[[5]]` is the object in car 5; `x[4:6]` is a train of cars 4-6."
>
> – @RLangTip

Because it can return only a single value, you must use `[[` with either a single positive integer or a string.

This works straightforwardly. Using the same 10-by-10 data frame from earlier, you can grab the contents of the 5th column with `df[[5]]`, `df[["V5"]]`, or `df$V5`. The `$` operator is nearly identical to the `[[` operator.[5]

There are nuances to the behavior of all of these different operators, but for they most part they aren't important – we've covered all the essential parts already. If you want to read more about the details, then consult section 3.2.1 in Wickham's *Advanced R*.

Supplemental exercises

Work through the following exercises thoroughly for some additional practice with names, lists, data frames, and subsetting.

Hint: You can modify objects by subsetting them and then using a standard assignment operator (`=`) to assign values to the subsets.

Lists

It is often useful to nest lists within each other repeatedly to store large amounts of heterogenous data in a convenient format. Indeed, the output of many modeling functions is just a named list of many different objects.

- Write a function `nesting_depth(L)` that takes as input a list `L` and returns the nesting depth of `L`. (For example, `nesting_depth(list(1, list(2, 3), list(4, 5)))` would return 2.)

In the following problems, let's call an n-domino a list with two integers, where both entries are integers from 0 to n inclusive. For example, `list(4,5)` is an n-domino (for any $n \geq 4$).

- Write a function to return a list of every unique n-domino, given n. Treat `list(a, b)` as being equivalent to `list(b, a)`. (*Hint:* If you have a list `L`, you can append an item to it by directly assigning something to its `(length(L)+1)`th position.)

[5] There's one minor exception. `x$y` is actually equivalent to `x[["y", exact = FALSE]]`, so `$` can partially match names (starting from the beginning of the string). For example, if `df` has a column named `"column"`, then `df$c` will return the output of `df$column`, assuming that no other columns in `df` have a name beginning with `"c"`.

  - It's very slow to continually append single items to lists over and over again, because you're copying over the structure with every iteration. If you can precalculate the number of list entries you'll need, you can *initialize* a list with `vector("list", list_size)`. (If possible, modify your code to do this and quantify the improvements in runtime using the `tictoc` package.)

A valid *circle* of n-dominoes is given by a list of n-dominoes, with the following properties:

- Given two consecutive dominoes `list(n, m)` and `list(p, q)`, where the latter domino is located immediately after the former domino in the circle of n-dominoes, we require that `m == p`.
- The 1st entry of the 1st n-domino is equal to the last entry of the last n-domino.

For example, `list(list(1, 2), list(2, 3), list(3, 1))` is a valid circle of n-dominoes.

- Write a function `is_circle(L)` that returns a logical value corresponding to whether or not `L` is a valid circle of n-dominoes.

Data frames

In the following, `mtcars` refers to a dataset that's loaded by default. These problems will begin with basic subsetting tasks and move to increasingly complex manipulations of data frames, many of which are tasks which will repeatedly occur in your data analyses.

Hint: Columns of a data frame can be removed by assigning `NULL` to them.

- With a single subset assignment command, change `x = 1:5` to be equivalent to `c(10, 11, 3, 4, 5)`.
- With a single subset assignment command, change `x = 1:10` to be equivalent to `c(1, 100, 3, 100, 5, 10, 7, 100, 9, 100)`. (*Hint:* You can subset with a logical vector.)[6]
- Why does `x = 1:5; x[NA]` yield five missing values? (*Hint:* How is it different from `NA_real_`?)
- Why does `mtcars[1:20]` return an error? How does it differ from the similar `mtcars[1:20, ]`?
- What does `df[is.na(df)] = 0` do? How does it work?

[6] A good way to do this is with `x[x %% 2 == 0] = rep(100, length(x[x %% 2 == 0]))`. We pass in the value of `length(...)` instead of `5` directly to improve the robustness of our code – our manual calculation of the value `5` could be incorrect.

- Let `x = c("a", "b", "a", "a", "b", "x", "b", "a")`. Construct a named vector called `fruits` such that the output of `fruits[x]` is equal to `c("apple", "banana", "apple", "apple", "banana", NA, "banana", "apple")`.
- Using `order()`, write a function to alphabetize the columns of a data frame by their names.
- Using `sample()`, write a function that takes a data frame as input and returns it with the order of its columns randomly permuted. After that, add a logical (boolean) flag to the function's parameters called `rows` defaulting to `FALSE` that permutes the rows as well if set to `TRUE`. (*I.e.,* calling `f(df)` would be equivalent to calling `f(df, rows=FALSE)` but `f(df, rows=TRUE)` would permute rows as well as columns.)
- Write a function that takes a data frame `df` and an integer `k` as input and returns `k` random columns of `df`, *sampled with replacement*.
- Write a function that takes a data frame `df` and an integer `m` as input and returns a random sample of `m` continuous rows of `df` as the output. (By continuous, we mean that you would return row `i`, row `i+1`, ... all the way to row `i+m-1` for some `i`.)
- Write a function that takes a data frame `df` and a string `colname` as input and returns a data frame without any columns that have name equal to the value of `colname`. There are many ways to do this, but you may find the expression `colname %in% names(df)` or the `match()` function useful. Try to do it multiple ways! (*Hint:* Don't forget about the edge case where multiple columns have identical names.)

Next, take a look at the built-in variable `letters`. We'll use `letters` to begin an exploration of R's string manipulation functions.

- Write a function that uses `grep()` to count the number of times each letter appears in the column names of an input data frame. It should return a numeric vector with appropriate names and of length 26 where the ith entry is the determined frequency of letter i.
- Write a function that uses `gsub()` to modify the column names of an input data frame by (1) changing every space (`" "`) into a dot (`"."`) and (2) appending `"_mod"` to the end of each name.
- Write a function that *removes* the last 4 characters of every column name of an input data frame. (If the name is 4 or fewer characters long, turn it into an empty string.) You may find `substr()` and `nchar()` helpful.
- Write a function that prints all of the row names of an input data frame joined together by an underscore (`"_"`) between each name. You may find `paste()` useful. (*Hint:* If you're struggling to use `do.call()`, read the documentation closely! You don't have to use `do.call()`, though.)

The following problem is a common programming interview question. Remember it well!

- Given a data frame of purely numeric data, write a function that returns the entries of the data frame ordered in a "spiral" fashion starting at the top left and proceeding counterclockwise and inward.
  - For example, the function appplied to `data.frame(matrix(1:9, nrow=3))` would return `c(1, 2, 3, 6, 9, 8, 7, 4, 5)`.
  - Add a `"clockwise"` parameter to your function, defaulting to `FALSE`, which if set to `TRUE` returns the entries corresponding to a clockwise traversal of the spiral.

We can also use these more complex data structures to facilitate our computational exploration of number-theoretic concepts.

- Think back to the exercise yesterday about the divisibility properties of Fibonacci numbers. Let F_i denote the ith Fibonacci number, starting with $F_1 = F_2 = 1$.
  - Make a data frame where the nth column is a logical vector with `TRUE` in position m if F_m divides F_n and `FALSE` otherwise. (The data frame can be as large as you want.)
  - Make another data frame in the same way, except look at whether or not m divides n (instead of F_m and F_n).
  - Explain yesterday's computational results using the patterns that you notice today.

Basic algorithms

We'll conclude with a selection of exercises about common algorithms. The material below is likely to show up on programming-focused interviews!

Run-length encoding

Run-length encoding is a simple form of data compression which represents data as a series of *runs* (sequences that consist of the same character repeated multiple times). It was originally used in the transmission of television signals and was used as an early form of image compression on CompuServe before the development of GIF. Indeed, the modern JPEG image compression algorithm incorporates run-length encoding into its functionality.

- Write a function `arg_max(v)` which takes in a numeric vector `v` and returns the *position* of its greatest element. If its greatest element occurs in

multiple places, print out the position of its first occurrence. You may find `max()` and `match()` helpful.

- Write a function `longest_run(v)` that prints out the longest "run" (sequence of consecutive identical values) in `v`. If there are multiple runs of the same length which quality, print out the first one. You may find `rle()` helpful.

Reservoir sampling

A classic task in data analysis is the problem of reading in n data items one by one for a very large and *unknown* n and choosing a random sample of k items. This can be done with reservoir sampling, introduced in 1985 by Jeffrey Vitter as "Algorithm R".

The algorithm consists of the following:

1. Initialize a "reservoir" of size k populated with the first k data items.
2. Continue reading in the data items. For the ith data element, generate a random integer j between 1 and i inclusive. If $j \leq k$, then the jth item in the reservoir is replaced with the ith data item.

Now, following the above description:

- Write a function `reservoir(v, k)` which iterates over the elements of `v` a *single time* and randomly chooses k of them with reservoir sampling. (For the random integer generation, combine `floor()` with `runif()`.)
- Run `reservoir()` repeatedly, choosing 5 elements randomly from a vector of 20 elements. For each item, calculate the probability of it being chosen for the sample.

Permutation generation

Quicksort and quickselect

One of the most straightforward sorting algorithms is *quicksort*, which sorts a list of length n in $O(n \log n)$ time. It was developed by Tony Hoare at Moscow State University as part of a translation project for the National Physical Laboratory requiring the alphabetical sorting of Russian words.

The steps of a simplified[7] form of the algorithm are as follows:

1. For a vector `L`, pick a random position `i`. The element `L[i]` is called the *pivot*. (If the pivot is the only element, return it.)

[7] The presented algorithm does not operate *in place*.

2. Form two vectors of elements `lesser` and `greater` which hold elements of `L` at positions *other than* `i` which are respectively lesser than or greater than `L[i]`. (Elements equal to `L[i]` can go in either one.)
3. Call the algorithm thus far `qs()`. Our result is the combination of concatenating together `qs(lesser)`, `L[i]`, and `qs(upper)`.

Now it's your turn:

- Implement a `quicksort(L)` function that sorts a vector of numbers `L` from least to greatest. Verify that your function works by writing a loop which generates 100 vectors of 10 random integers and compares the output of `quicksort()` to the built-in `sort()`. Compare the performance of `quicksort()` to that of `sort()`.

The *quickselect* algorithm, which is similar to quicksort, allows you to find the kth largest (or smallest) element of a list of n elements in $O(n)$ time. The difference in the algorithms is that in each iteration, we only have to recurse into *one* of the two subdivisions of the vector, because we can tell which one holds our desired value based on the value of k and the sizes of `lesser` and `greater`.

- Implement a `quickselect(L, k)` function which finds the kth smallest element of L.