Documents & Links

Something fun for July 1 ¶

(Subtitle: "Some basic computer science." This reading is entirely optional.)

We've talked about a number of handy concepts now. The first is functions and their local variables, which don't "step on" any other variable that might exist when the function is called. This makes the execution of a function it's "own little world."

```
equalish <- function(a, b, epsilon = 0.00001) {
  res <- abs(a - b) < epsilon
  return(res)
}

res <- "A message."
print(equalish(4, 4.00001))  # prints TRUE
print(res)                   # prints "A message.
```

Second, lists, which can store data elements of various types, *including other lists*:

```
a_list <- list("A", "B", "C", list("X", "Y"))
```

And if-statements, which allow for conditionally-executing code:

```
if(length(a_list) < 10) {
  print("a_list is less than 10 elements")
} else {
  print("a_list has 10 or more elements")
}
```

As mentioned in class, we can put these concepts together to do some fairly remarkable things.

Nesting Lists

Lists are at the very least a straightforward way to store a set of data elements in order:

```
chars <- list("A", "C", "D", "F")
```

Suppose, hypothetically speaking, that lists were more limited, such that they could only store exactly 2 elements. How might we store this sequence of data? One way to do so would be to use the first "slot" of a list for a data element, and the second for a sublist–we could make this nesting structure as deep as we like to store lots of elements. For the last sublist we can use `NULL` as a placeholder to indicate that no sublist exists there. (Which is slightly different than setting a list element to `NULL` with `<-` to delete it.)

```
chars <- list("A", NULL)                    # storing one element
chars <- list("A", list("C", NULL))         # storing two elements
chars <- list("A", list("C", list("D", list("F", NULL))))  # storing 4 elements
```

Obviously, this isn't the the most convenient way to store data (or is it?). For example, to get the third element, we need to say something like `third_el <- chars[[2]][[2]][[1]]` (or do we?).

What if we wanted to print all of the elements in order? This is where functions and if-statements come into play, in a clever (and really fun, in my opinion) strategy known as *recursion*. Our strategy will be to write a function

called `print_list()`, that will take as a parameter `l` a nested list of this kind. The function will extract the first element from the list (the first data element, `"A"` above) and print it. Then it will extract the second data element–the sublist holding the rest of the elements. And what will it do with nested sublist? Well, the problem repeats itself: we want to print the elements of it in order. So it will call `print_list()` on it! Of course, it should only do this if the second element is *not* `NULL`.

```
print_list <- function(l) {
  data_el <- l[[1]]
  sublist <- l[[2]]
  print(data_el)
  if(!is.null(sublist)) {
    print_list(sublist)
  }
}

print_list(chars)
```

The output:

```
[1] "A"
[1] "C"
[1] "D"
[1] "F"
```

This remarkable strategy–a function that calls itself–works because of local variables. In the function, `l`, `data_el`, and `sublist` are all local, and so multiple calls can all work on their own versions of these variables without interfering with each other. It's also important to note that the "size" of the problem is reduced with every function call–the `l` parameter is one sublist shorter each time, so eventually the process runs out of data (when `is.null(sublist)` returns `TRUE`).

Question: What do you think would happen if the `print(data_el)` *were moved to the bottom of the function? Try it and see if your guess was correct.*

We can use a similar strategy to get the Nth element of a list: we'll write a function called `return_nth()` that takes a nested list `l` and an index `n`. If `n == 1`, then the function can just return the first element of `l`. If not, it can extract the sublist, and call `return_nth()` but using `n-1` (e.g., the 10th element of a list is the 9th element of the sublist).

```
return_nth <- function(l, n) {
  if(n == 1) {
    return(l[[1]])
  } else {
    sublist <- l[[2]]
    newindex <- n - 1
    answer <- return_nth(sublist, newindex)
    return(answer)
  }
}

print(return_nth(chars, 3))  # prints "D"
```

Question: see if you can modify the above such that if the user requests a number `n` *larger than the length of the list, an* `NA` *value is returned rather than producing an error.*

One more example of the concept to drive the point home: adding a new element to the end of the list. In this case we'll write a function `append_end()` that takes a nested list `l` and a new element `new_element`. If `l`'s sublist is `NULL`, then we can create a new nested list structure with `l`'s first element and a sublist with `list(new_element,`

`NULL`).

On the other hand, if `l`'s sublist is not `NULL`, then we'll extract the sublist and have the new element appended to that, before returning a new nesting list with the first element and the result.

```
append_end <- function(l, new_element) {
  if(is.null(l[[2]])) {
    first_el <- l[[1]]
    newlist <- list(first_el, list(new_element, NULL))
    return(newlist)
  } else {
    first_el <- l[[1]]
    sublist <- l[[2]]
    newsublist <- append_end(sublist, new_element)
    newlist <- list(first_el, newsublist)
    return(newlist)
  }
}

chars_with_x <- append_end(chars, "X")
print_list(chars_with_x)
```

The output for the above:

```
[1] "A"
[1] "C"
[1] "D"
[1] "F"
[1] "X"
```

In the parlance of recursion, the simple case above above is called the "base case"–it's the easiest and smallest version of the problem that could be solved. The other case (where the function calls itself) is called the "recursive" case. Most recursive functions have these two cases (see the `return_nth()` function above), and they serve as a good starting point for writing these kinds of functions.

Question: see if you can write a recursive function that returns the number of elements of a nested list.

Trees

One task that frequently arises is keeping a set of elements in order as new elements are added. This is actually possible with the nested list structure above with a recursive function, but it requires a little more trickery.

Instead, let's direct our trickery differently. Suppose that our lists are now slightly more advanced: rather than storing exactly two elements, they can store three. Now we can have our data element in the middle and two sublists, one on either side of the data element:

```
chars <- list(NULL, "C", NULL)                                  # 1 element
chars <- list(list(NULL, "A", NULL), "C", list(NULL, "F", NULL)) # 3 elements
```

If we want to ensure that the list stays in order, we can simply enforce that all the elements in the "left" sublist are less than the element, which is in turn less than all the elements in the "right" sublist.

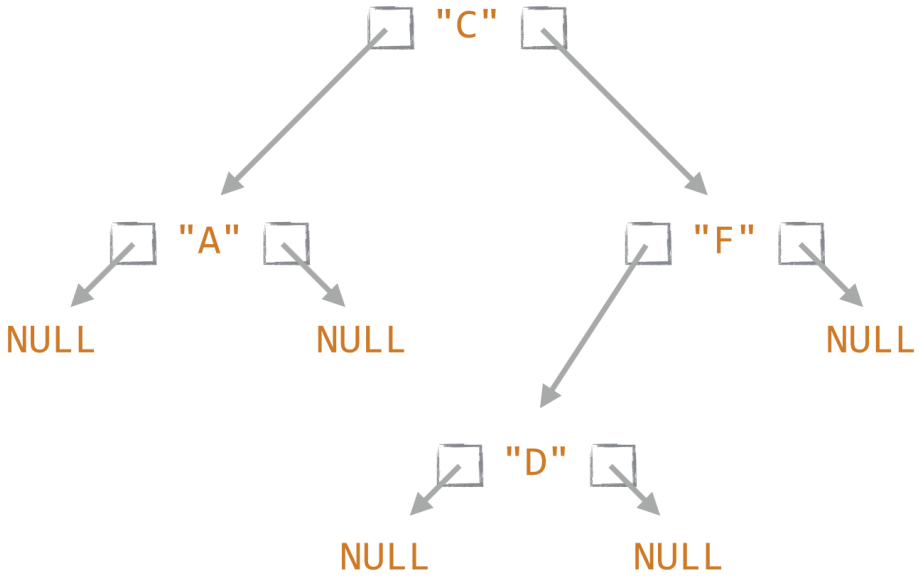
This means that there are a variety of ways we could organize the 3 data elements `"A"`, `"C"`, and `"F"`:

```
chars <- list(NULL, "A", list(NULL, "C", list(NULL, "F", NULL)))
chars <- list(list(list(NULL, "A", NULL), "C", NULL), "F", NULL)
```

Actually, for larger sets, the number of ways of organizing them is very large. Here's a list with four elements:

```
c <- list(list(NULL, "A", NULL), "C", list(list(NULL, "D", NULL), "F", NULL))
```

This is getting unwieldy very quickly. Actually, we can better visualize this structure as a "tree" (traditionally drawn upside down).



How might we print these elements in order? In the earlier example, the function extracted the first element, printed it, and then extracted the sublist and called a function to print that (so long as it wasn't `NULL`). We can do something analogous here: we'll write a `print_tree()` function that takes a nested tree `t`: it will extract the left subtree, the element, and the right subtree. If the left subtree isn't `NULL`, it will call for that to be printed (causing all "lesser" elements to be printed in order), then it will print the element, then if the right subtree isn't `NULL` it will call for that to be printed (causing all "greater" elements to be printed in order).

```
print_tree <- function(t) {
  left <- t[[1]]
  element <- t[[2]]
  right <- t[[3]]

  if(!is.null(left)) {
    print_tree(left)
  }

  print(element)

  if(!is.null(right)) {
    print_tree(right)
  }
}

print_tree(c)
```

The output is faithfully `"A"`, then `"C"`, then `"D"`, and finally `"F"`.

Question: what happens if `print(element)` *is moved to different locations within the function? Can you get the elements to be printed in reverse order?*

What is really fascinating is that it is fairly easy to add new elements to the tree such that everything stays in the

correct right order; we'll do this with a function called `insert_into_tree()` that takes a tree `t` and a `new_element`. The strategy works much like for the earlier append function: the new element will either go in the left subtree, or the right subtree, depending on whether it is smaller than or greater than the element at the current node (starting at the top). Let's say it is smaller: if the left subtree is `NULL`, then the left subtree can become `list(NULL, new_element, NULL)`. If not, then `insert_into_tree()` can be called on the left subtree, and the answer used as the new left subtree. Same goes for the right.

```
insert_into_tree <- function(t, new_element) {
  left <- t[[1]]
  element <- t[[2]]
  right <- t[[3]]

  if(new_element < element) {
    if(is.null(left)) {
      newleft <- list(NULL, new_element, NULL)
      newtree <- list(newleft, element, right)
      return(newtree)
    } else {
      newleft <- insert_into_tree(left, new_element)
      newtree <- list(newleft, element, right)
      return(newtree)
    }
  } else {
    if(is.null(right)) {
      newright <- list(NULL, new_element, NULL)
      newtree <- list(left, element, newright)
      return(newtree)
    } else {
      newright <- insert_into_tree(right, new_element)
      newtree <- list(left, element, newright)
      return(newtree)
    }
  }
}

c <- insert_into_tree(c, "B")
print_tree(c)
```

This code could be done in fewer lines, but I've purposely broken out the four cases to illustrate them clearly. The output of printing after the addition of `"B"`:

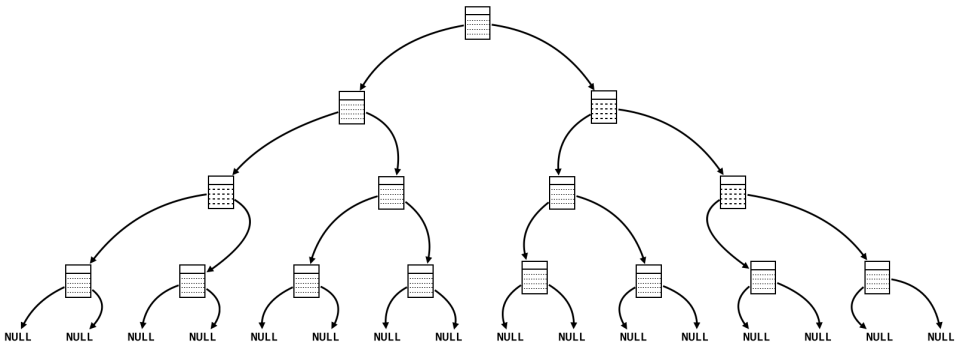
```
[1] "A"
[1] "B"
[1] "C"
[1] "D"
[1] "F"
```

Where did the `"B"` node go in the tree diagram above? Well, it would have gone left (and down) of `"C"`, and then right (and down) of `"A"`. Thus, new nodes are always added at the "leaves" of the tree, where `NULL` values are.

Questions: write a recursive function that returns the number of elements in the tree. Write one that returns the first element. Write one that returns the last element.

All of this might seem like a tedious way to simply store a set of elements in order, but this solution has some interesting properties. Consider the problem of inserting a new element in the right location. If the data were stored as a simple array, then inserting a new first element would require shifting all of the other elements over one; if there are 1,000,000 elements, this would be roughly 1,000,000 steps of work.

On the other hand, inserting an element into the tree essentially follows a path from the top node (also called the root) to the location where it is inserted. The amount of work is equal to the length of this path. If the tree is nice and bushy (a.k.a. balanced):



Then the length of any path from the top to the bottom is fairly short compared to the total number of nodes; in fact if there are *n* nodes then in a nice bushy tree the path lengths are *log2(n)*. In the example of a tree with 1,000,000 elements, the amount of work is roughly only about 20 steps. This data structure is known as a binary tree, whereas the simpler nested list above is known as a linked list.

Reading for June 26

Common vector functions

As vectors (specifically numeric vectors) are so ubiquitous, R has dozens (hundreds, actually) of functions that do useful things with them. While we can't cover all of them, we can quickly cover a few that will be important in future chapters.

First, we've already seen the `seq()`, and `length()` functions; the former generates a numeric vector comprising a sequence of numbers, and the latter returns the length of a vector as a single-element integer vector.

```
range <- seq(0, 7, 0.2)               # 0.0 0.2 0.4 ... 7.0
len_range <- length(range)            # 36
```

Presented without example, `mean()`, `sd()`, and `median()` return the mean, standard deviation, and median of a numeric vector, respectively. (Provided that none of the input elements are `NA`, though all three accept the `na.rm = TRUE` parameter.) Generalizing `median()`, the `quantile()` function returns the Yth percentile of a function, or multiple percentiles if the second argument has more than one element.

```
quantiles_range <- quantile(range, c(0.25, 0.5, 0.75))
print(quantiles_range)
```

The output is a named numeric vector:

```
 25%  50%  75%
1.75 3.50 5.25
```

The `unique()` function removes duplicates in a vector, leaving the remaining elements in order of their first occurence, and the `rev()` function reverses a vector.

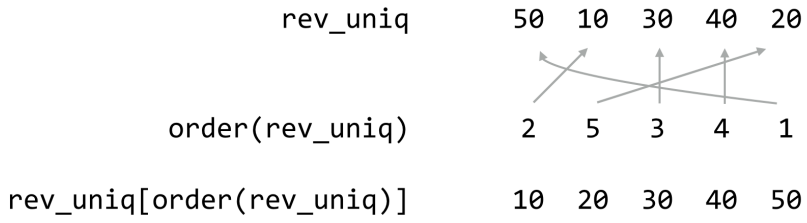
```
values <- c(20, 40, 30, 20, 10, 50, 10)
values_uniq <- unique(values)         # 20 40 30 10 50

rev_uniq <- rev(values_uniq)          # 50 10 30 40 20
```

There is a `sort()` function which simply sorts a vector (in natural order for numerics and integers, and lexicographic order for character vectors). Perhaps more interesting is the `order()` function, which returns an integer vector of indices describing where the original elements of the vector would need to be placed to produce a sorted order.

```
order_rev_uniq <- order(rev_uniq)     # 2 5 3 4 1
```

In this example, the order vector, `2 5 3 4 1` indicates that the second element of `rev_uniq` would come first, followed by the fifth, and so on. Thus, we could produce a sorted version of `rev_uniq` with `rev_uniq[order_rev_uniq]` (by virtue of index-based selection), or more succinctly with `rev_uniq[order(rev_uniq)]`.



Importantly, this allows us to re-arrange multiple vectors with a common order determined by a single one. For example, given two vectors, `id` and `score`, we might decide to rearrange both sets in alphabetical order for `id`.

```
id <- c("cc4", "aa6", "bb3")
score <- c(20.05, 35.62, 42.71)

id_sorted <- id[order(id)]
score_sorted <- score[order(id)]

print(id_sorted)                    # [1] "aa6" "bb3" "cc4"
print(score_sorted)                 # [1] 35.62 42.71 20.05
```

The `sample()` function returns a random sampling from a vector of a given size, either with replacement or without as specified with the `replace =` parameter (`FALSE` is the default if unspecified).

```
values <- c(5, 10, 15, 20, 25, 30)

sample_1 <- sample(values, 3, replace = FALSE)        # 15 5 30
sample_2 <- sample(values, 3, replace = TRUE)         # 15 30 15
```

The `rep()` function repeats a vector to produce a longer vector. We can repeat in an element-by-element fashion, or over the whole vector, depending on whether the `each =` parameter is used or not.

```
count <- c(1, 2)

count_rep1 <- rep(count, 3)                           # 1 2 1 2 1 2
count_rep2 <- rep(count, each = 3)                    # 1 1 1 2 2 2
```

Last (but not least) for this section is the `is.na()` function: given a vector with elements that are possibly `NA` values, returns a logical vector whole elements are `TRUE` in indices where the original was `NA`. This allows us to easily indicate which elements of vectors are `NA` and remove them.

```
values_char <- c("5.7", "4.3", "a9b3", "2.4")
values <- as.numeric(values_char)                    # 5.7 4.3 NA 2.4

values_na <- is.na(values)                           # FALSE FALSE TRUE FALSE

values_no_nas <- values[!values_na]                  # 5.7 4.3 2.4
# OR
values_no_nas <- values[!is.na(values_na)]           # 5.7 4.3 2.4
```

Notice the use of `!` in the above to negate logical vector returned by `is.na()`.

Generating random data

R excels at working with probability distributions, including generating random samples from them. Many distributions are supported, including the Normal (Gaussian), Log-Normal, Exponential, Gamma, Student's t, and so on. Here we'll just look at generating samples from a few for use in future examples.

First, the `rnorm()` function generates a numeric vector of a given length sampled from the Normal distribution with specified mean (with `mean =`) and standard deviation (with `sd =`).

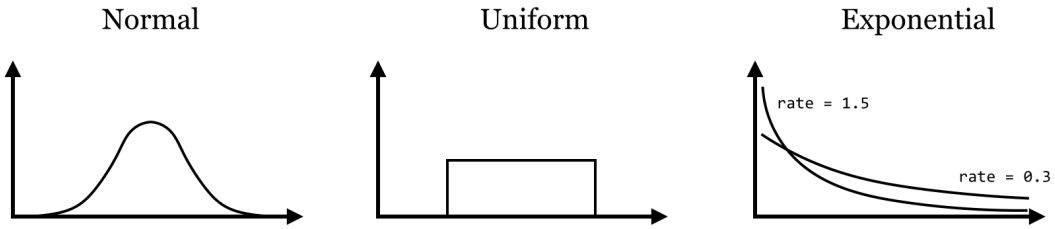
```
sample_norm <- rnorm(5, mean = 6, sd = 2)      # e.g. 7.07 2.4 4.5 6.2 5.1
```

Similarly, the `runif()` function samples from a uniform distribution limited by a minimum and maximum value.

```
sample_unif <- runif(5, min = 2, max = 6)      # e.g. 2.1 4.06 2.48 4.67 5.80
```

And, the `rexp()` generates data from an Exponential distribution with a given "rate" parameter, controlling the rate of decay of the density function (the mean of large samples will approach `1.0/rate`).

```
sample_exp <- rexp(5, rate = 1.5)              # e.g. 0.24 0.50 0.01 0.55 0.30
```



R includes a large number of statistical tests, though we won't be coving much in the way of statistics other than a few driving examples. The `t.test()` function runs two-sided Student's T-test comparing the means of two vectors. What is returned is a more complex datatype with class `"htest"`.

```
sample_1 <- rnorm(100, mean = 10, sd = 4)
sample_2 <- rnorm(100, mean = 12, sd = 4)

ttest_result <- t.test(sample_1, sample_2)
print(class(ttest_result))                     # [1] "htest"
print(ttest_result)
```

When printed, this complex datatype formats itself into nicely human-readable output:

```
Welch Two Sample t-test

data:  sample_1 and sample_2
t = -2.6847, df = 193.503, p-value = 0.007889
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -2.690577 -0.411598
sample estimates:
mean of x mean of y
 10.03711  11.58819
```

Random Info

- The State of Naming Conventions in R Rasmus Bååth, R Journal 4/2, 2012.

PDFs

- Syllabus
- Flyer