Big Data in Economics

Lecture 9: Functions in R: (1) Introductory concepts

Grant R. McDermott

University of Oregon | EC 510

Contents

Note: This will be the first of several lectures on programming. Today, we'll cover the basics of function writing and functional programming (especially as it applies to iteration). My goal is to provide a solid foundation through a bunch of conceptually simple examples, which will allow us to tackle more complex problems later on.

Software requirements

R packages

- New: **pbapply**
- Already used: **tidyverse**, **data.table**

We'll mostly be using the tidyverse to access the **purrr** package. Note that we're also going to upgrade our data.table installation to the latest development version, which provides some new features that are not yet (as of the time of writing) available in the CRAN release. Let's install (if necessary) and load everything:

```
if (!require("data.table")) install.packages("data.table")
data.table::update.dev.pkg()
```

```
## R data.table package is up-to-date at dd7609e83132e19c9be80d71c73e8a8f95e19b27 (1.12.9)
```

```
if (!require("pacman")) install.packages("pacman")
pacman::p_load(pbapply, data.table, tidyverse)
```

> **Tip:** If for some reason you have trouble installing the development version of data.table, see here. We'll also only be using a function or two from it today, so don't worry too much about upgrading if you are having difficulty.

Basic syntax

We have already seen and used a multitude of functions in R. Some of these functions come pre-packaged with base R (e.g. `mean()`), while others are from external packages (e.g. `dplyr::filter()`). Regardless of where they come from, functions in R all adopt the same basic syntax:

```
function_name(ARGUMENTS)
```

For much of the time, we will rely on functions that other people have written for us. However, you can — and should! — write your own functions too. This is easy to do with the generic **`function()`** function.[1] The syntax will again look familiar to you:

```
function(ARGUMENTS) {
  OPERATIONS
  return(VALUE)
}
```

While it's possible and reasonably common to write anonymous functions like the above, we typically write functions because we want to reuse code. For this typical use-case it makes sense to name our functions.[2]

```
my_func <-
  function(ARGUMENTS) {
    OPERATIONS
    return(VALUE)
  }
```

For some short functions, you don't need to invoke the curly brackets or assign an explicit return object (more on this below). In these cases, you can just write your function on a single line:

```
my_short_func <- function(ARGUMENTS) OPERATION
```

Try to give your functions short, pithy names that are informative to both you and anyone else reading your code. This is harder than it sounds, but will pay off down the road.

A simple example

Let's write out a simple example function, which gives the square of an input number.

```
square <-       ## Our function name
  function(x) { ## The argument(s) that our function takes as an input
    x^2         ## The operation(s) that our function performs
  }
```

Test it.

```
square(3)
```

```
## [1] 9
```

Great, it works. Note that for this simple example we could have written everything on a single line; i.e. `square ← function(x) x^2` would work just as well. (Confirm this for yourself.) However, we're about to add some extra conditions and options to our function, which will strongly favour the multi-line format.

Aside: I want to stress that our new `square()` *function is not particularly exciting... or, indeed, useful. R's built-in arithmetic functions already take care of (vectorised) exponentiation and do so very efficiently. (See* `?Arithmetic`*.) However, we're going to continue with this conceptually simple example, since it will provide a clear framework for demonstrating some general principles about functions in R.*

Specifying return values

Notice that we didn't specify a return value for our function. This will work in many cases because R's default behaviour is to automatically return the final object that you created within the function. However, this won't always be the case. I thus advise that you get into the habit of assigning the return object(s) explicitly. Let's modify our function to do exactly that.

[1] Yes, it's a function that let's you write functions. Very meta.
[2] Remember: "In R, everything is an object and everything has a name."

```
square <-
  function(x) {
    x_sq <- x^2   ## Create an intermediary object (that will be returned)
    return(x_sq)  ## The value(s) or object(s) that we want returned.
  }
```

Again, test that it works.

```
square(5)
```

```
## [1] 25
```

Specifying an explicit return value is also helpful when we want to return more than one object. For example, let's say that we want to remind our user what variable they used as an argument in our function:

```
square <-
  function(x) { ## The argument(s) that our function takes as an input
    x_sq <- x^2 ## The operation(s) that our function performs
    return(list(value=x, value_squared=x_sq)) ## The list of object(s) that we want returned.
  }
```

```
square(3)
```

```
## $value
## [1] 3
##
## $value_squared
## [1] 9
```

Note that multiple return objects have to be combined in a list. I didn't have to name these separate list elements — i.e. "value" and "value_squared" — but it will be helpful for users of our function. Nevertheless, remember that many objects in R contain multiple elements (vectors, data frames, and lists are all good examples of this). So we can also specify one of these "array"-type objects within the function itself if that provides a more convenient form of output. For example, we could combine the input and output values into a data frame:

```
square <-
  function(x) {
    x_sq <- x^2
    df <- tibble(value=x, value_squared=x_sq) ## Bundle up our input and output values into a convenient datafra
    return(df)
  }
```

Test.

```
square(12)
```

```
## # A tibble: 1 x 2
##   value value_squared
##   <dbl>         <dbl>
## 1    12           144
```

Specifying default argument values

Another thing worth noting about R functions is that you can assign default argument values. You have already encountered some examples of this in action.[3] We can add a default option to our own function pretty easily.

```
square <-
  function(x = 1) { ## Setting the default argument value
```

[3] E.g. Type `?rnorm` and see that it provides a default mean and standard deviation of 0 and 1, respectively.

```
    x_sq <- x^2
    df <- tibble(value=x, value_squared=x_sq)
    return(df)
  }
```

```
square() ## Will take the default value of 1 since we didn't provide an alternative.
```

```
## # A tibble: 1 x 2
##   value value_squared
##   <dbl>         <dbl>
## 1     1             1
```

```
square(2) ## Now takes the explicit value that we give it.
```

```
## # A tibble: 1 x 2
##   value value_squared
##   <dbl>         <dbl>
## 1     2             4
```

We'll return the issue of specifying default values (and handling invalid inputs) in the next lecture on function debugging.

Aside: Environments and lexical scoping

Before continuing, I want to highlight the fact that none of the intermediate objects that we created within the above functions (`x_sq`, `df`, etc.) have made their way into our global environment. Take a moment to confirm this for yourself by looking in the "Environment" pane of your RStudio session.

R has a set of so-called *lexical scoping* rules, which govern where it stores and evaluates the values of different objects. Without going into too much depth, the practical implication of lexical scoping is that functions operate in a quasi-sandboxed *environment*. They don't return or use objects in the global environment unless they are forced to (e.g. with a `return()` command). Similarly, a function will only look to outside environments (e.g. a level "up") to find an object if it doesn't see the object named within itself.

We'll explore the ideas of separate environments and lexical scoping a bit further when we get to the Functional programming section below. We'll also go into more depth during the next lecture on debugging.

Control flow

Now that we've got a good sense of the basic function syntax, it's time to learn control flow. That is, we want to control the order (or "flow") of statements and operations that our functions evaluate.

if and ifelse

We've already encountered conditional statements like `if()` and `ifelse()` at various point in the course thus far. However, let's see how they can work in our own bespoke functions by slightly modifying our previous square function. This time, instead of specifying a default input value of 1 in the function argument itself, we'll specify a value of `NULL`. Then we'll use an `if()` statement to reassign this default to one.

```
square <-
  function(x = NULL) { ## Default value of NULL
    if (is.null(x)) x=1 ## Re-assign default to 1
    x_sq <- x^2
    df <- tibble(value=x, value_squared=x_sq)
    return(df)
  }
square()
```

```
## # A tibble: 1 x 2
```

```
##   value value_squared
##   <dbl>         <dbl>
## 1     1             1
```

Why go through the rigmarole of specifying a NULL default inpute if we're going to change it to 1 anyway? Admittedly, this is a pretty silly thing to do in the above example. However, consider what it buys us in the next code chunk:

```
square <-
  function(x = NULL) {
    if (is.null(x)) { ## Start multiline if statement with `{`
      x=1
      message("No input value provided. Using default value of 1.") ## Message to users
      } ## Close multiline if statement with `}`
    x_sq <- x^2
    df <- tibble(value=x, value_squared=x_sq)
    return(df)
  }
square()
```

```
## No input value provided. Using default value of 1.
```

```
## # A tibble: 1 x 2
##   value value_squared
##   <dbl>         <dbl>
## 1     1             1
```

This time, by specifying NULL in the argument — alongside the expanded `if()` statement — our function now both takes a default value *and* generates a helpful message.[4] Note too the use of curly brackets for conditional operations that span multiple lines after an `if()` statement. This provides a nice segue to `ifelse()` statements. As we've already seen , these be written as a single conditional call where the format is:

```
ifelse(CONDITION, DO IF TRUE, DO IF FALSE)
```

Within our own functions, though we're more likely to write them over several lines. Consider, for example a new function that evaluates whether our `square()` function is doing its job properly.

```
eval_square <-
  function(x) {
    if (square(x)$value_squared == x*x) { ## condition
      ## What to do if the condition is TRUE
      message("Nailed it.")
    } else {
      ## What to do if the condition is FALSE
      message("Dude, your function sucks.")
    }
  }
eval_square(64)
```

```
## Nailed it.
```

Aside: ifelse gotchas and alternatives The base R `ifelse()` function normally works great and I use it all the time. However, there are a couple of "gotcha" cases that you should be aware of. Consider the following (silly) function which is designed to return either today's date, or the day before.[5]

[4] Think about how you might have tried to achieve this if we'd assigned the `x = 1` default directly in function argument as before. It quickly gets complicated, because how can your message discriminate whether a user left the argument blank or deliberately entered `square(1)`?

[5] The dots (`...`) argument is a convenient way to allow for unspecified arguments to be used in our functions. This is beyond the scope of our current lecture, but can prove to be an incredibly useful and flexible programming strategy. I highly encourage you to look at the relevant section in *Advanced R* to get a better idea.

```
today <- function( ... ) ifelse( ... , Sys.Date(), Sys.Date()-1)
today(TRUE)
```

```
## [1] 18388
```

You are no doubt surprised to find that our function returns a number instead of a date. This is because `ifelse()` automatically converts date objects to numeric as a way to get around some other type conversion strictures. Confirm for yourself by converting it back the other way around with: `as.Date(today(TRUE), origin = "1970-01-01")`.

To guard against this type of unexpected behaviour, as well as incorporate some other optimisations, both the tidyverse (through **dplyr**) and **data.table** offer their own versions of ifelse statements. I won't explain these next code chunks in depth (consult the relevant help pages if needed), but here are adapated versions of our `today()` function based on these alternatives.

First, `dplyr::if_else()`:

```
today2 <- function( ... ) dplyr::if_else( ... , Sys.Date(), Sys.Date()-1)
today2(TRUE)
```

```
## [1] "2020-05-06"
```

Second, `data.table::fifelse()`:

```
today3 <- function( ... ) data.table::fifelse( ... , Sys.Date(), Sys.Date()-1)
today3(TRUE)
```

```
## [1] "2020-05-06"
```

case when (nested ifelse)

As you may have guessed, it's certainly possible to write nested `ifelse()` statements. For example,

```
ifelse(CONDITION1, DO IF TRUE, ifelse(CONDITION2, DO IF TRUE, ifelse( ... )))
```

However, these nested statements quickly become difficult to read and troubleshoot. A better solution was originally developed in SQL with the `CASE WHEN` statement. Both **dplyr** with `case_when()` and **data.table** with `fcase()` provide implementations R.[6] Here is a simple illustration of both implementations.

```
x = 1:10
## dplyr::case_when()
case_when(
  x <= 3 ~ "small",
  x <= 7 ~ "medium",
  TRUE ~ "big" ## Default value. Could also write `x > 7 ~ "big"` here.
  )
```

```
##  [1] "small"  "small"  "small"  "medium" "medium" "medium" "medium" "big"
##  [9] "big"    "big"
```

```
## data.table::fcase()
fcase(
    x <= 3, "small",
    x <= 7, "medium",
    default = "big" ## Default value. Could also write `x > 7, "big"` here.
    )
```

```
##  [1] "small"  "small"  "small"  "medium" "medium" "medium" "medium" "big"
##  [9] "big"    "big"
```

[6] Note that `data.table::fcase()` is only available in the latest development version at present.

Not to belabour the point, but you can easily use these *case when* implementations inside of data frames/tables too.

```
## dplyr::case_when()
tibble(x = 1:10) %>%
    mutate(grp = case_when(x <= 3 ~ "small",
                           x <= 7 ~ "medium",
                           TRUE ~ "big"))
```

```
## # A tibble: 10 x 2
##        x grp
##    <int> <chr>
##  1     1 small
##  2     2 small
##  3     3 small
##  4     4 medium
##  5     5 medium
##  6     6 medium
##  7     7 medium
##  8     8 big
##  9     9 big
## 10    10 big
```

```
## data.table::fcase()
data.table(x = 1:10)[, grp := fcase(x <= 3, "small",
                                    x <= 7, "medium",
                                    default = "big")][]
```

```
##       x    grp
##  1:   1  small
##  2:   2  small
##  3:   3  small
##  4:   4 medium
##  5:   5 medium
##  6:   6 medium
##  7:   7 medium
##  8:   8    big
##  9:   9    big
## 10:  10    big
```

Iteration

Alongside control flow, the most important early programming skill to master is iteration. In particular, we want to write functions that can iterate — or *map* — over a set of inputs.[7] By far the most common way to iterate across different programming languages is *for* loops. Indeed, we already saw some examples of *for* loops back in the shell lecture. However, while R certainly accepts standard *for* loops, I'm going to advocate that you adopt what is known as a "functional programming" approach to writing loops. Let's dive into the reasons why and how these approaches differ.

Vectorisation

The first question you should be asking yourself is: "Do I need to iterate at all?"

You may remember from a previous lecture that I spoke about R being *vectorised*. Which is to say that you can apply a function to every element of a vector at once, rather than one at a time. Let's demonstrate this property with our `square` function:

[7] Our focus today will only be on sequential iteration, but we'll return to parallel iteration in the lecture after next.

```
square(1:5)
```

```
## # A tibble: 5 x 2
##   value value_squared
##   <int>         <dbl>
## 1     1             1
## 2     2             4
## 3     3             9
## 4     4            16
## 5     5            25
```

```
square(c(2, 4))
```

```
## # A tibble: 2 x 2
##   value value_squared
##   <dbl>         <dbl>
## 1     2             4
## 2     4            16
```

So you may not need to worry about explicit iteration at all. (Bonus: Vectorised operations are also extremely fast.) That being said, there are certainly cases where you will need to iterate. Let's explore with some simple examples that will provide a mental springboard for thinking about more complex cases.

Note: Most of the iteration examples that follow could themselves be vectorised. But my goal is to illustrate some general principles about writing iterative functions in R (what format they take, etc.), rather than optimising performance of these toy examples.

for loops

In R, standard *for* loops take a pretty intuitive form. For example:

```
for(i in 1:10) print(LETTERS[i])
```

```
## [1] "A"
## [1] "B"
## [1] "C"
## [1] "D"
## [1] "E"
## [1] "F"
## [1] "G"
## [1] "H"
## [1] "I"
## [1] "J"
```

Note that in cases where we want to "grow" an object via a *for* loop, we first have to create an empty (or NULL) object.

```
kelvin <- 300:305
fahrenheit <- NULL
# fahrenheit <- vector("double", length(kelvin)) ## Better than the above. Why?
for(k in 1:length(kelvin)) {
  fahrenheit[k] <- kelvin[k] * 9/5 - 459.67
}
fahrenheit
```

```
## [1] 80.33 82.13 83.93 85.73 87.53 89.33
```

Unfortunately, basic *for* loops in R also come with some downsides. Historically, they used to be significantly slower and memory consumptive than alternative methods (see below). This has largely been resolved, but I've still run into cases

where an inconspicuous *for* loop has brought an entire analysis crashing to its knees.[8] The bigger problem with *for* loops, however, is that they deviate from the norms and best practices of **functional programming**.

Functional programming

The concept of functional programming (FP) is arguably the most important thing that you can take away from today's lecture. In his excellent book, *Advanced R*, Hadley Wickham explains the core idea as follows.

> R, at its heart, is a functional programming (FP) language. This means that it provides many tools for the creation and manipulation of functions. In particular, R has what's known as first class functions. You can do anything with functions that you can do with vectors: you can assign them to variables, store them in lists, pass them as arguments to other functions, create them inside functions, and even return them as the result of a function.

That may seem a little abstract, so here is video of Hadley giving a much more intuitive explanation through a series of examples.

Summary: *for* loops tend to emphasise the *objects* that we're working with (say, a vector of numbers) rather than the *operations* that we want to apply to them (say, get the mean or median or whatever). This is inefficient because it requires us to continually write out the *for* loops by hand rather than getting an R function to create the for-loop for us.

As a corollary, *for* loops also pollute our global environment with the variables that are used as counting variables. Take a look at your "Environment" pane in RStudio. What do you see? In addition to the `kelvin` and `fahrenheit` vectors that we created, we also see two variables `i` and `k` (equal to the last value of their respective loops). Creating these auxiliary variables is almost certainly not an intended outcome when your write a for-loop.[9] More worringly, they can cause programming errors when we inadvertently refer to a similarly-named variable elsewhere in our script. So we best remove them manually as soon as we're finished with a loop.

```
rm(i,k)
```

Another annoyance arrived in cases where we want to "grow" an object as we iterate over it (e.g. the `fahrenheit` object in our second example). In order to do this with a *for* loop, we had to go through the rigmarole of creating an empty object first.

FP allows to avoid the explicit use of loop constructs and its associated downsides. In practice, there are two ways to implement FP in R:

1. The `*apply` family of functions in base R.
2. The `map*()` family of functions from the **purrr**.

Let's explore these in more depth.

1) lapply and co. Base R contains a very useful family of `*apply` functions. I won't go through all of these here — see `?apply` or this blog post among numerous excellent resources — but they all follow a similar philosophy and syntax. The good news is that this syntax very closely mimics the syntax of basic for-loops. For example, consider the code below, which is analgous to our first *for* loop above, but now invokes a **`base::lapply()`** call instead.

```
# for(i in 1:10) print(LETTERS[i]) ## Our original for loop (for comparison)
lapply(1:10, function(i) LETTERS[i])
```

```
## [[1]]
## [1] "A"
##
## [[2]]
## [1] "B"
##
## [[3]]
```

[8] Exhibit A. Trust me: debugging these cases is not much fun.

[9] The best case I can think of is when you are trying to keep track of the number of loops, but even then there are much better ways of doing this.

```
## [1] "C"
##
## [[4]]
## [1] "D"
##
## [[5]]
## [1] "E"
##
## [[6]]
## [1] "F"
##
## [[7]]
## [1] "G"
##
## [[8]]
## [1] "H"
##
## [[9]]
## [1] "I"
##
## [[10]]
## [1] "J"
```

A couple of things to notice.

First, check your "Environment" pane in RStudio. Do you see an object called "i" in the Global Environment? (The answer should be"no".) Again, this is because of R's lexical scoping rules, which mean that any object created and invoked by a function is evaluated in a sandboxed environment outside of your global environment.

Second, notice how little the basic syntax changed when switching over from `for()` to `lapply()`. Yes, there are some differences, but the essential structure remains the same: We first provide the iteration list (`1:10`) and then specify the desired function or operation (`LETTERS[i]`).

Third, notice that the returned object is a *list*. The `lapply()` function can take various input types as arguments — vectors, data frames, lists — but always returns a list, where each element of the returned list is the result from one iteration of the loop. (So now you know where the "l" in "**l**apply" comes from.)

Okay, but what if you don't want the output in list form? There several options here.[10] However, the method that I use most commonly is to bind the different list elements into a single data frame with either `dplyr::bind_rows()` or `data.table::rbindlist()`. For example, here's a a slightly modified version of our function that now yields a data frame:

```
# library(tidyverse) ## Already loaded

lapply(1:10, function(i) {
  df <- tibble(num = i, let = LETTERS[i])
  return(df)
  }) %>%
  bind_rows()
```

```
## # A tibble: 10 x 2
##      num let
##    <int> <chr>
##  1     1 A
##  2     2 B
##  3     3 C
```

[10] For example, we could pipe the output to `unlist()` if you wanted a vector. Or you could use use `sapply()` instead, which I'll cover shortly.

```
##  4     4 D
##  5     5 E
##  6     6 F
##  7     7 G
##  8     8 H
##  9     9 I
## 10    10 J
```

Taking a step back, while the default list-return behaviour may not sound ideal at first, I've found that I use `lapply()` more frequently than any of the other `apply` family members. A key reason is that my functions normally return multiple objects of different type (which makes lists the only sensible format)… or a single data frame (which is where `dplyr::bind_rows()` or `data.table::rbindlist()` come in).

Aside: Quick look at sapply() Another option that would work well in the this particular case is `sapply()`, which stands for "**s**implify apply". This is essentially a wrapper around `lapply` that tries to return simplified output that matches the input type. If you feed the function a vector, it will try to return a vector, etc.

```
sapply(1:10, function(i) LETTERS[i])
```

```
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J"
```

Aside: Progress bars! Who doesn't like progress bars? Personally, I find it incredibly helpful to see how a function is progressing, or get a sense of how much longer I can expect to wait before completion.

Along those lines, I'm a big fan of the **pbapply** package, which is a lightweight wrapper around the `*apply` functions that adds a progress bar. `pbapply` offers versions for all of the `*apply` family, but the one that I use the most is (unsurprisingly) `pbapply::pblapply()`.

Note: You will need to run this next example interactively to see the effect properly.

```
# library(pbapply) ## Already loaded

pblapply(1:10, function(i) {
  df <- tibble(num = i, let = LETTERS[i])
  Sys.sleep(1)
  return(df)
  }) %>%
  bind_rows()
```

```
## # A tibble: 10 x 2
##      num let
##    <int> <chr>
##  1     1 A
##  2     2 B
##  3     3 C
##  4     4 D
##  5     5 E
##  6     6 F
##  7     7 G
##  8     8 H
##  9     9 I
## 10    10 J
```

Another thing that I really like about the `pblapply()` function is that it allows for easy implementation of parallel (i.e. multicore) processing that works across operating systems. We'll cover this next week, though.

There's also a newish package on the scene, **progressr**, which provides a unified API for progress updates across multiple iteration frameworks in R. I won't cover it here, but it's pretty neat and simple to use, so check it out.

2) purrr package The tidyverse offers its own enhanced implementation of the base `*apply()` functions through the **purrr** package.[11] The key function to remember here is `purrr::map()`. And, indeed, the syntax and output of this command are effectively identical to `base::lapply()`:

```
# lapply(1:10, num_to_alpha) ## lapply version from earlier
map(1:10, num_to_alpha)
```

```
## [[1]]
## # A tibble: 1 x 2
##     num let
##   <int> <chr>
## 1     1 A
##
## [[2]]
## # A tibble: 1 x 2
##     num let
##   <int> <chr>
## 1     2 B
##
## [[3]]
## # A tibble: 1 x 2
##     num let
##   <int> <chr>
## 1     3 C
##
## [[4]]
## # A tibble: 1 x 2
##     num let
##   <int> <chr>
## 1     4 D
##
## [[5]]
## # A tibble: 1 x 2
##     num let
##   <int> <chr>
## 1     5 E
##
## [[6]]
## # A tibble: 1 x 2
##     num let
##   <int> <chr>
## 1     6 F
##
## [[7]]
## # A tibble: 1 x 2
##     num let
##   <int> <chr>
## 1     7 G
##
## [[8]]
## # A tibble: 1 x 2
##     num let
##   <int> <chr>
```

[11] In their words: "The apply family of functions in base R solve a similar problem *[i.e. to purrr]*, but purrr is more consistent and thus is easier to learn."

```
## 1     8 H
##
## [[9]]
## # A tibble: 1 x 2
##     num let
##   <int> <chr>
## 1     9 I
##
## [[10]]
## # A tibble: 1 x 2
##     num let
##   <int> <chr>
## 1    10 J
```

Given these similarities, I won't spend much time on **purrr**. Although, I do think it will be the optimal entry point for many you when it comes to programming and iteration. You have already learned the syntax, so it should be very easy to switch over. However, one additional thing I wanted to flag for today is that `map()` also comes with its own variants, which are useful for returning objects of a desired type. For example, we can use `purrr::map_df()` to return a data frame.

```
# lapply(1:10, num_to_alpha) %>% bind_rows() ## lapply version from earlier.
map_df(1:10, num_to_alpha)
```

```
## # A tibble: 10 x 2
##      num let
##    <int> <chr>
##  1     1 A
##  2     2 B
##  3     3 C
##  4     4 D
##  5     5 E
##  6     6 F
##  7     7 G
##  8     8 H
##  9     9 I
## 10    10 J
```

Note that this is more efficient (i.e. involves less typing) than the `lapply()` version, since we don't have to go through the extra step of binding rows at the end.

Create and iterate over named functions

As you may have guessed already, we can split the function and the iteration (and binding) into separate steps. This is generally a good idea, since you typically create (named) functions with the goal of reusing them.

```
## Create a named function
num_to_alpha <-
  function(i) {
  df <- tibble(num = i, let = LETTERS[i])
  return(df)
  }
```

Now, we can easily iterate over our function using different input values. For example,

```
lapply(1:10, num_to_alpha) %>% bind_rows()
```

```
## # A tibble: 10 x 2
##      num let
##    <int> <chr>
```

```
##  1     1 A
##  2     2 B
##  3     3 C
##  4     4 D
##  5     5 E
##  6     6 F
##  7     7 G
##  8     8 H
##  9     9 I
## 10    10 J
```

Or,

```
lapply(c(1, 5, 26, 3), num_to_alpha) %>% bind_rows()
```

```
## # A tibble: 4 x 2
##     num let
##   <dbl> <chr>
## 1     1 A
## 2     5 E
## 3    26 Z
## 4     3 C
```

Iterate over multiple inputs

Thus far, we have only been working with functions that take a single input when iterating. For example, we feed them a single vector (even though that vector contains many elements that drive the iteration process). But what if we want to iterate over multiple inputs? Consider the following function, which takes two separate variables `x` and `y` as inputs, combines them in a data frame, and then uses them to create a third variable `z`.

Note: Again, this is a rather silly function that we could easily improve upon using standard (vectorised) tools. But my goal here is to demonstrate programming principles with simple examples that carry over to more complicated cases where vectorisation is not possible.

```
## Create a named function
multi_func <-
  function(x, y) {
  df <-
    tibble(x = x, y = y) %>%
    mutate(z = (x + y)/sqrt(x))
  return(df)
  }
```

Before continuing, quickly test that it works using non-iterated inputs.

```
multi_func(1, 6)
```

```
## # A tibble: 1 x 3
##       x     y     z
##   <dbl> <dbl> <dbl>
## 1     1     6     7
```

Great, it works. Now let's imagine that we want to iterate over various levels of both `x` and `y`. There are two basics approaches that we can follow to achieve this:

1. Use `base::mapply()` or `purrr::pmap()`.
2. Use a data frame of input combinations.

I'll quickly review both approaches, continuing with the `multi_func()` function that we just created above.

1) Use `mapply()` or `pmap()` Both base R — through `mapply()` — and purrr — through `pmap` — can handle multiple input cases for iteration. The latter is easier to work with in my opinion, since the syntax is closer (nearly identical) to the single input case. Still, I'll demonstrate using both versions below.

First, `base::mapply()`:

```
## Note that the inputs are now moved to the *end* of the call.
## mapply is based on sapply, so we also have to tell it not to simplify if we want to keep the list structure.
mapply(
  multi_func,
  1:5, ## Our "x" vector input
  y=6:10, ## Our "y" vector input
  SIMPLIFY = F ## Tell it not to simplify to keep the list structure
  ) %>%
  bind_rows()
```

```
## # A tibble: 5 x 3
##       x     y     z
##   <int> <int> <dbl>
## 1     1     6  7
## 2     2     7  6.36
## 3     3     8  6.35
## 4     4     9  6.5
## 5     5    10  6.71
```

Second, `purrr::pmap()`:

```
## Note that the inputs are combined in a list.
pmap_df(list(x=1:5, y=6:10), multi_func)
```

```
## # A tibble: 5 x 3
##       x     y     z
##   <int> <int> <dbl>
## 1     1     6  7
## 2     2     7  6.36
## 3     3     8  6.35
## 4     4     9  6.5
## 5     5    10  6.71
```

2) Using a data frame of input combinations While the above approaches work perfectly well, I find that I don't actually use either all that much in practice. Rather, I prefer to "cheat" by feeding multi-input functions a *single* data frame that specifies the necessary combination of variables by row. I'll demonstrate how this works in a second, but first let me explain why.

It basically boils down to the fact that I feel this gives me more control over my functions and inputs.

- I don't have to worry about accidentally feeding separate inputs of different lengths. Try running the above functions with an `x` vector input of `1:10`, for example. (Leave everything else unchanged.) `pmap()` will at least fail to iterate and give you a helpful message, but `mapply` will actually complete with totally misaligned columns. Putting everything in a (rectangular) data frame forces you to ensure the equal length of inputs *a priori*.
- Similarly, I often need to run a function over all possible combinations of different inputs. A really convenient way to do this is with the `base::expand.grid()` function, which automatically generates a data frame of all combinations.[12] So it's convenient for me to use this data frame as an input directly in my function.

[12] We encountered some related tidyverse functions (`tidyr::expand()` and `tidyr::complete()`) in an earlier lecture, as well as your second assignment. Of course, you could could also extract the columns of the data frame as separate vectors and then feed these into `pmap()`/`mapply()` if you prefer.

- In my view, it's just simpler and cleaner to keep things down to a single input. This will be harder to see in the simple example that I'm going to present next, but I've found that it can make a big difference with complicated functions that have a lot of nesting (i.e. functions of functions) and/or parallelization.

Those justifications aside, let's see how this might work with an example. Consider the following function:

```
parent_func <-
  ## Main function: Takes a single data frame as an input
  function(input_df) {
    df <-
      ## Nested iteration function
      map_df(
      1:nrow(input_df), ## i.e. Iterate (map) over each row of the input data frame
      function(n) {
        ## Extract the `x` and `y` values from row "n" of the data frame
        x <- input_df$x[n]
        y <- input_df$y[n]
        ## Use the extracted values
        df <- multi_func(x, y)
        return(df)
      })
    return(df)
    }
```

There are three conceptual steps to the above code chunk:

1. First, I create a new function called `parent_func()`, which takes a single input: a data frame containing `x` and `y` columns (and potentially other columns too).
2. This input data frame is then passed to a second (nested) function, which will *iterate over the rows of the data frame*.
3. During each iteration, the `x` and `y` values for that row are passed to our original `multi_func()` function. This will return a data frame containing the desired output.

Let's test that it worked using two different input data frames.

```
## Case 1: Iterate over x=1:5 and y=6:10
input_df1 <- tibble(x=1:5, y=6:10)
parent_func(input_df1)
```

```
## # A tibble: 5 x 3
##       x     y     z
##   <int> <int> <dbl>
## 1     1     6  7
## 2     2     7  6.36
## 3     3     8  6.35
## 4     4     9  6.5
## 5     5    10  6.71
```

```
## Case 2: Iterate over *all possible combinations* of x=1:5 and y=6:10
input_df2 <- expand.grid(x=1:5, y=6:10)
# input_df2 <- expand(input_df1, x, y) ## Also works
parent_func(input_df2)
```

```
## # A tibble: 25 x 3
##        x     y     z
##    <int> <int> <dbl>
##  1     1     6  7
##  2     2     6  5.66
##  3     3     6  5.20
```

```
##  4     4     6  5
##  5     5     6  4.92
##  6     1     7  8
##  7     2     7  6.36
##  8     3     7  5.77
##  9     4     7  5.5
## 10     5     7  5.37
## # ... with 15 more rows
```

Further resources

In the next two lectures, we'll dive into more advanced programming and function topics (debugging, parallel implementation, etc.). However, I hope that today has given you solid grasp of the fundamentals. I highly encourage you to start writing some of your own functions. You will be doing this a *lot* as your career progresses. Establishing an early mastery of function writing will put you on the road to awesome data science successTM. Here are some additional resources for both inspiration and reference:

- Garrett Grolemund and Hadley Wickham's *R for Data Science* book — esp. chapters 19 ("Functions)") and 21 ("Iteration)") — covers much of the same ground as we have here, with particular emphasis on the **purrr** package for iteration.
- If you're looking for an in-depth treatment, then I can highly recommend Hadley's *Advanced R* (2nd ed.) He provides a detailed yet readable overview of all the concepts that we touched on today, including more on his (and R's) philosophy regarding functional programming (see Section ||).
- If you're in the market for a more concise overview of the different `*apply()` functions, then I recommend this blog post by Neil Saunders.
- On the other end of the scale, Jenny Bryan (all hail) has created a fairly epic purrr tutorial mini-website. (Bonus: She goes into more depth about working with lists and list columns.)