Functional programming

EDH7916 | Spring 2020

Benjamin Skinner

This lesson covers two core programming processes: control flow and writing functions.

Like other programming languages, R reads scripts from top to bottom, implementing each command or function as it comes. But like other languages, it also has special functions that you can use to change how R moves through the script — to *control the flow* of the analysis process. With these special functions, you can selectively implement and repeat sections of code.

R also allows you to write your own functions. While most of your data analyses will use functions from base R or a library you load, you may find it useful to write your own functions from time to time. You might use your *user-written functions* to perform specialized tasks or simply as wrappers for sections of code that you otherwise might repeat multiple times.

This lesson is divided into three sections. In the first section, we'll go through some common control flow processes. In the second section, we'll go over the process for building our own functions. In the last section, we'll work through two examples of how functional programming can streamline a data analysis workflow.

But before moving on, we'll quickly talk about the guiding principle behind this lesson.

DRY vs WET programming

The watchwords for this lesson are **DRY** vs **WET**:

- **DRY**: *Don't repeat yourself*
- **WET**: *Write every time*

Let's say you have a three-step analysis process for 20 files (read, lower names, add a column). Under a *WET* programming paradigm in which each command gets its own line of code, that's 60 lines of code. If the number of your files grows to 50, that's now 150 lines of code — for just three tasks! When you write every time, you not only make your code longer and harder to parse, you also *increase* the likelihood that your code will contain bugs while simultaneously *decreasing* its scalability.

If you need to repeat an analytic task (which may be a set of commands), then it's better to have one statement of that process that you repeat, perhaps in a loop or in a function. Don't repeat yourself — say it once and have R repeat it for you!

The goal of DRY programming is not abstraction or slickness for its own sake. That runs counter to the clarity and replicability we've been working toward. Instead, we aspire to DRY code since it is more scalable and less buggy than WET code. To be clear, a function or loop can still have bugs, but the bugs it introduces are often *the same across repetitions* and fixed *at a single point of error*. That is, it's typically easier to *debug* when the bug has a single root cause than when it could be anywhere in 150 similar but *slightly* different lines of code.

As we work through the lesson examples, keep in the back of your mind:

1. *What would this code look like if I wrote everything twice (WET)*?
2. *How does this DRY process not only reduce the number of lines of code, but also make my intent clearer?*

Setup

We'll use a combination of nonce data and the school test score data we've used in a past lesson. We won't read in the school test score data until the last section, but we'll continue following our good organizational practice by setting the directory paths at the top of our script.

```
## ---------------------------
## libraries
## ---------------------------

library(tidyverse)
```

```
── Attaching packages ─────────────────────────────────────── tidyverse 1.3.0 ──

✔ ggplot2 3.2.1     ✔ purrr   0.3.3
✔ tibble  2.1.3     ✔ dplyr   0.8.4
✔ tidyr   1.0.2     ✔ stringr 1.4.0
✔ readr   1.3.1     ✔ forcats 0.4.0

── Conflicts ────────────────────────────────────────── tidyverse_conflicts() ──
✖ dplyr::filter() masks stats::filter()
✖ dplyr::lag()    masks stats::lag()
```

NB: As we have done in the past few lessons, we'll run this script assuming that our working directory is set to the `scripts` directory.

```
## ---------------------------
## directory paths
## ---------------------------

## assume we're running this script from the ./scripts subdirectory
dat_dir <- file.path("..", "data")
sch_dir <- file.path(dat_dir, "sch_test")
bys_dir <- file.path(sch_dir, "by_school")
```

Part 1: Control flow

As stated above, by *control flow*, I simply mean the functions that help you change how your script is read. Repeating commands often involves a *loop*, which is what it sounds like: upon reaching a loop, R will repeat the code inside the loop (*looping back up the beginning of the section*) for a certain number of times or until some condition is met. Once completed, R will go back to reading each line in order like normal.

If you google around, you may find that loops have a bad reputation in R, mostly for being slow. But they aren't *that* slow and they are easy to write and understand.

for

The `for()` function allows you to build the aptly named **for loops**. There are few ways to use `for()`, but its construction is the same: `for (variable in sequence)`.

Reading it backwards, the `sequence` is just the set of numbers or objects that we're going to work through. The `variable` is a new variable that will temporarily hold a value from the sequence in each run through the loop. When the `sequence` is finished, so is the loop.

First, let's loop through a sequence of 10 numbers, printing each one at a time as we work through the loop.

```
## make vector of numbers between 1 and 10
num_sequence <- 1:10

## loop through, printing each num_sequence value, one at a time
for (i in num_sequence) {
    print(i)
}
```

```
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
```

Notice the braces `{}` that come after `for()`. This is the code in the loop that will be repeated as long as the loop is run. With each loop, `i` takes on the next value in the `num_sequence`. This is why we see 1 through 10 printed to the console.

Let's do it again, but this time with characters.

```
## character vector using letters object from R base
chr_sequence <- letters[1:10]

## loop through, printing each chr_sequence value, one at a time
for (i in chr_sequence) {
    print(i)
}
```

```
[1] "a"
[1] "b"
[1] "c"
[1] "d"
[1] "e"
[1] "f"
[1] "g"
[1] "h"
[1] "i"
[1] "j"
```

Once more, with each loop, `i` takes on each `chr_sequence` value in turn and `print()` prints it to the console.

> **Quick exercise** Can you modify the above loop so that it works through both the `num_sequence` and `chr_sequence` in the same loop? (HINT: how might you combine/concatenate the two sequences?)

Another way to make a for loop is work through a vector by its indices, that is, instead of pulling out each item directly and storing it in `i`, we can use `i` as an *index counter* and then call items based on their index: `chr_sequence[i]`. Here's an example.

```
## for loop by indices
for (i in 1:length(chr_sequence)) {
    print(chr_sequence[i])
}
```

```
[1] "a"
[1] "b"
[1] "c"
[1] "d"
[1] "e"
[1] "f"
[1] "g"
[1] "h"
[1] "i"
[1] "j"
```

To understand what's happening, let's break the code into pieces to make it clearer.

Inside the `for()` parentheses, we have `i in 1:length(chr_sequence)`. We know what `i in` means since it's like what we've seen before. What's `1:length(chr_sequence)`? First, it looks like the colon construction we've seen before, `<start>:<end>`, which means give the full sequence of values from `<start>` through `<end>`.

Since we made it above, we know that there are ten letters in `chr_sequence`. We could just use `1:10`. However, this violates our **no magic numbers** rule (what happens if we want to add or subtract letters from the list later?).

We can get around this issue by using the base-R function, `length()`, which will return the number of items in a one-dimensional object. Since we know that `length(chr_sequence)` is equal to `10`, that means that `1:length(chr_sequence)` is the same thing as saying `1:10`. It's just another more flexible way to get the end number of our sequence. We can show this by just printing `i` again.

```
## for loop by indices (just show indices)
for (i in 1:length(chr_sequence)) {
    print(i)
}
```

```
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
```

Back to the original function, we see that inside the braces (`{}`), we have `print(chr_sequence[i])`. From earlier lessons, we know that brackets (`[]`) are way of pulling out specific values from a vector. We've only used numbers before, but we can also use variables that represent numbers. Here's a non-looped test:

```
## confirm that we can use variables as indices
i <- 1                    # set i == 1
chr_sequence[i]
```

```
[1] "a"
```

```
i <- 2                    # now set i == 2
chr_sequence[i]           # notice that code is exactly the same here
```

```
[1] "b"
```

We know that `i` is going to take on values 1 through 10 in the loop, which means the `print()` function will get `chr_sequence[1]`, `chr_sequence[2]`, and so on. Because of the brackets, these will turn into…`a`, `b`, and so on. We should get the same thing as before! Let's put all the pieces together and run again:

```
## for loop by indices (once again)
for (i in 1:length(chr_sequence)) {
    print(chr_sequence[i])
}
```

```
[1] "a"
[1] "b"
[1] "c"
[1] "d"
[1] "e"
[1] "f"
[1] "g"
[1] "h"
[1] "i"
[1] "j"
```

Success!

Whether you decide to loop using actual values from the sequence or indices will usually depend on the code you want to run in the loop. Sometimes one way works better and other times the other. Just do whatever works best for you at that time.

> **Quick exercise** Add another print statement to the last loop that shows the value of `i` with each loop.

while

The `while()` function is similar to `for()` except that it doesn't have a predetermined stopping point. As long the expression inside the parentheses is `TRUE`, the loop will keep going. Only when it becomes `FALSE` will it stop.

One way to use a `while()` loop is to set up a counter. When the counter reaches some value, the expression inside the `while()` parentheses is no longer true and the loop stops.

```
## set up a counter
i <- 1
## with each loop, add one to i
while(i < 11) {
    print(i)
    i <- i + 1
}
```

```
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
```

```
[1] 8
[1] 9
[1] 10
```

Using a `while()` loop with a counter is often the same as using a `for()` loop with a sequence. If that's the case, it's probably better just to use a `for()` loop.

`while()` loops are most useful when it's not clear from the start when the loop should stop. Imagine you have an algorithm that should only stop when a certain number is reached. If the time it takes to reach the number changes depending on the input, then a `for()` loop probably won't work, but a `while()` loop will.

You have to careful, however, with `while()` loops. If you forget to increment the counter (like I did the first time I set up this example), the loop won't ever stop because `i` will never get larger and will always be less than 11! If your `while()` loop will only stop when a certain condition is met, it's still a good idea to build in a pre-specified number of trials. If your loop has tried X times to meet the condition and still hasn't done so, it should stop with an error or return what it has so far (depending on your needs).

You have been warned!

if

We've already used a version of if, `ifelse()`, quite a bit. We can use **if** statements in our script to decide whether a section of code should be run or skipped. We can also use `if()` inside a `for()` loop to set a condition that changes behavior on some iterations of the loop.

```
## only print if number is not 5
for (i in num_sequence) {
    if (i != 5) {
        print(i)
    }
}
```

```
[1] 1
[1] 2
[1] 3
[1] 4
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
```

Notice how `5` wasn't printed to the console. It worked!

Quick exercise Change the condition to print only numbers below 3 and above 7.

We can add one or more `else if() / else()` partners to `if()` if we need, for example, option **B** to happen if option **A** does not.

```
## if/else loop
for (i in num_sequence) {
    if (i != 3 & i != 5) {
        print(i)
    } else if (i == 3) {
        print('three')
    } else {
        print('five')
```

```
    }
}
```

```
[1] 1
[1] 2
[1] "three"
[1] 4
[1] "five"
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
```

As with {dplyr} verbs, the small number of control functions (and there are some others) can be combined in an infinite number of ways to help you control how your script is read. If you find, however, that your script keeps getting more complex, with multiple nested loops and `ifelse()` exceptions, it might be time to either rethink your approach or to write your own functions that can handle expectations in a clearer way.

Part 2: Writing functions

You can write your own functions in R and should! They don't need to be complex. In fact, they tend to be best when kept simple. Mostly, you want a function to do one thing really well.

To make a function, you use the `function()` function. Put the code that you want your function to run in the braces `{}`. Any arguments that you want your function to take should be in the parentheses `()` right after the word `function`. The name of your function is the name of the object you assign it to.

Let's make one. The function below, `say_hi()`, doesn't take any arguments and prints a simple string when called. After you've built it, call your function using its name, not forgetting to include the parentheses.

```
## function to say hi!
say_hi <- function() {
    print("Hi!")
}

## call it
say_hi()
```

```
[1] "Hi!"
```

Let's make another one with an argument so that it's more flexible. Let's have our function take a name and print it. We'll use the base-R function, `paste0()`, to combine all the string parts into one string. `paste0()` is like `paste()`, but it assumes we don't want any space between the pieces. That works for us here because we'll add our spaces manually.

```
## function to say hi!
say_hi <- function(name) {
    ## combine (notice we add space after comma)
    out_string <- paste0("Hi, ", name, "!")
    ## print output string
    print(out_string)
}

## call it
say_hi("Leo")
```

```
[1] "Hi, Leo!"
```

This time, we want it to print out a sequence of numbers, but we want to be able to change the number each time we call it.

```
## new function to print sequence of numbers
print_nums <- function(num_vector) {
    ## this code looks familiar...
    for (i in num_vector) {
        print(i)
    }
}

## try it out!
print_nums(1:10)
```

```
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
```

Notice how the variable `num_vector` is repeated in both the main function argument and inside the `for` parentheses. The `for()` function sees `num_vector` and looks for it in the main function. It finds it because the `num_vector` you give the main function, `print_nums()`, is passed through to the code inside. Now `for()` can see it and use it! Let's try a few more inputs:

```
## v1
print_nums(1)
```

```
[1] 1
```

```
## v2
print_nums(1:5)
```

```
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

```
## v3
print_nums(seq(1, 20, by = 2))
```

```
[1] 1
[1] 3
[1] 5
[1] 7
[1] 9
[1] 11
[1] 13
[1] 15
[1] 17
```

```
[1] 19
```

> **Quick exercise** What happens if you forget to put an argument in your new function? How do you think you might set a default argument for `num_vector`? Could you set it equal to something?

One last thing to keep in mind about functions is that they follow Vegas rules: *what happens inside the function, stays inside the function.* A more technical way of stating this is that the function has its own **scope** and objects created / processes run within that scope stay within that scope. That's why even though we created an object called `out_string` in our `say_hi()` function, we can't call it directly from the console. It only exists while the function is running and goes away once the function completes.

Even if we repeat an object name that exists in our **global** (working) environment inside our function, the value we give it inside the function will override the global value within the function. But once the function is finished, the global value will again take precedence. Don't worry to much about scoping (here's more information if you are interested), but just be aware that it's generally good practice that if you want an object for use inside your function, you should either create it there or make it an argument.

Part 3: Practical examples

Example 1: missing data

Now that we've seen some control flow and programming methods, let's move to a more realistic use case. In this first example, we'll make a function that fills in missing values, a common task we've had. First, we'll generate some fake data with missing values.

Note that since we're using R's `sample()` function, your data will look a little different from mine due to randomness in the sample, but everything will work the same.

```
## create a data frame with around 10% missing values (-97,-98,-99) in
## three columns
df <- tibble("id" = 1:100,
             "age" = sample(c(seq(11,20,1), -97),
                            size = 100,
                            replace = TRUE,
                            prob = c(rep(.09, 10), .1)),
             "sibage" = sample(c(seq(5,12,1), -98),
                               size = 100,
                               replace = TRUE,
                               prob = c(rep(.115, 8), .08)),
             "parage" = sample(c(seq(45,55,1), c(-98,-99)),
                               size = 100,
                               replace = TRUE,
                               prob = c(rep(.085, 11), c(.12, .12)))
             )
## show
df
```

```
# A tibble: 100 x 4
      id   age sibage parage
   <int> <dbl>  <dbl>  <dbl>
 1     1    16     11     46
 2     2    18     12     55
 3     3    19      5     47
 4     4    19      7     55
```

```
 5     5    14     11     49
 6     6    18      9     55
 7     7    14      6     52
 8     8    11      8     48
 9     9    11    -98     53
10    10    20      7     51
# … with 90 more rows
```

We could fix these manually like we have been in past lessons and assignments, but it would be nice have a shorthand function. The function needs to flexible though, because the missing data values are coded differently in each column.

```
## function to fix missing values
fix_missing <- function(x, miss_val) {
    ## use ifelse(< test >, < do this if TRUE >, < do that if FALSE >)
    x <- ifelse(x %in% miss_val,        # is x == any value in miss_val?
                NA,                     # TRUE: replace with NA
                x)                      # FALSE: return original value as is
    ## return corrected x
    return(x)
}
```

Our `fix_missing()` function should meet our needs. It takes the same `ifelse()` function we've used before, but instead of using the name of the object (like `df`), uses an argument name `x` that we can set each time. It does the same for `miss_val`. Instead of choosing a hard-coded value (a magic number), we can change it each time we call the function. Let's try it out.

```
## check
df %>%
    count(age)
```

```
# A tibble: 11 x 2
     age     n
   <dbl> <int>
 1   -97     5
 2    11    10
 3    12     9
 4    13     8
 5    14     5
 6    15     9
 7    16    10
 8    17    13
 9    18    13
10    19     9
11    20     9
```

```
## missing values in age are coded as -97
df <- df %>%
    mutate(age = fix_missing(age, -97))

## recheck
df %>%
    count(age)
```

```
# A tibble: 11 x 2
     age     n
   <dbl> <int>
```

```
 1    11    10
 2    12     9
 3    13     8
 4    14     5
 5    15     9
 6    16    10
 7    17    13
 8    18    13
 9    19     9
10    20     9
11    NA     5
```

It worked! All the values that were -97 before, are now in the `NA` table column. Importantly, none of the other values changed.

> **Quick exercise** Correct the missing values in the other columns using our new function. **NOTE** that `parage` has missing values of `-98` and `-99`.

Example 2: batch read files

Download all files

In our lesson on appending, joining, and merging, we read in a few administrative test score files. In that lesson, we read in each file individually and then appended them.

```
## read in all Bend Gate test score files
df_1 <- read_csv(file.path(bys_dir, "bend_gate_1980.csv"))
df_2 <- read_csv(file.path(bys_dir, "bend_gate_1981.csv"))
df_3 <- read_csv(file.path(bys_dir, "bend_gate_1982.csv"))
df_4 <- read_csv(file.path(bys_dir, "bend_gate_1983.csv"))
df_5 <- read_csv(file.path(bys_dir, "bend_gate_1984.csv"))
df_6 <- read_csv(file.path(bys_dir, "bend_gate_1985.csv"))

## append
df <- bind_rows(df_1, df_2, df_3, df_4, df_5, df_6)

## show
df
```

```
# A tibble: 6 x 5
  school     year  math  read science
  <chr>     <dbl> <dbl> <dbl>   <dbl>
1 Bend Gate  1980   515   281     808
2 Bend Gate  1981   503   312     814
3 Bend Gate  1982   514   316     816
4 Bend Gate  1983   491   276     793
5 Bend Gate  1984   502   310     788
6 Bend Gate  1985   488   280     789
```

That was fine then, but when you see that much repeated code, you should immediately consider using a more functional programming approach. What if we wanted to read in all the files for the other schools as well? That would be a total of 24 very similar lines of code — ripe for introducing bugs.

Let's combine a number of the skills we've learned to this point — functions, regular expressions, and pipes — to read and bind all the test score files in a more functional (and less buggy) manner.

First, we'll store the names of the files in an object using `list.files()`. We've used the `list.files()` function a few times in class when checking that we were in the correct working directory. In those situations, we've just printed the output to the console. But as with all things R, we can save that output in an object instead and put it to good use.

```
## get names (with full path) of all school test score files
files <- list.files(bys_dir, full.names = TRUE)

## show
files
```

```
 [1] "../data/sch_test/by_school/bend_gate_1980.csv"
 [2] "../data/sch_test/by_school/bend_gate_1981.csv"
 [3] "../data/sch_test/by_school/bend_gate_1982.csv"
 [4] "../data/sch_test/by_school/bend_gate_1983.csv"
 [5] "../data/sch_test/by_school/bend_gate_1984.csv"
 [6] "../data/sch_test/by_school/bend_gate_1985.csv"
 [7] "../data/sch_test/by_school/east_heights_1980.csv"
 [8] "../data/sch_test/by_school/east_heights_1981.csv"
 [9] "../data/sch_test/by_school/east_heights_1982.csv"
[10] "../data/sch_test/by_school/east_heights_1983.csv"
[11] "../data/sch_test/by_school/east_heights_1984.csv"
[12] "../data/sch_test/by_school/east_heights_1985.csv"
[13] "../data/sch_test/by_school/niagara_1980.csv"
[14] "../data/sch_test/by_school/niagara_1981.csv"
[15] "../data/sch_test/by_school/niagara_1982.csv"
[16] "../data/sch_test/by_school/niagara_1983.csv"
[17] "../data/sch_test/by_school/niagara_1984.csv"
[18] "../data/sch_test/by_school/niagara_1985.csv"
[19] "../data/sch_test/by_school/spottsville_1980.csv"
[20] "../data/sch_test/by_school/spottsville_1981.csv"
[21] "../data/sch_test/by_school/spottsville_1982.csv"
[22] "../data/sch_test/by_school/spottsville_1983.csv"
[23] "../data/sch_test/by_school/spottsville_1984.csv"
[24] "../data/sch_test/by_school/spottsville_1985.csv"
```

> **Quick exercise** Rerun `list.files()` again, but set the `full.names` argument to the default value of `FALSE`. What does the output look like? Why is this a problem? (Once finished, be sure to set `full.names = TRUE` again and rerun.)

Now that we have an object, we can read in the files using a loop. We'll need something to store them in along the way. We'll use a blank list. We've not used lists before, but think of them as special vectors, which we have used.

Once we have a list, we'll read in each file, but instead of storing it in an object like `df`, we'll put it in the list.

```
## init list
df_list <- list()

## use loop to read in files
for (i in 1:length(files)) {
    ## read in file (f) and store in list (note double brackets for list)
    df_list[[i]] <- read_csv(files[i])
}
```

```
## show first 3 items
df_list[1:3]
```

```
[[1]]
# A tibble: 1 x 5
  school     year  math  read science
  <chr>     <dbl> <dbl> <dbl>   <dbl>
1 Bend Gate  1980   515   281     808

[[2]]
# A tibble: 1 x 5
  school     year  math  read science
  <chr>     <dbl> <dbl> <dbl>   <dbl>
1 Bend Gate  1981   503   312     814

[[3]]
# A tibble: 1 x 5
  school     year  math  read science
  <chr>     <dbl> <dbl> <dbl>   <dbl>
1 Bend Gate  1982   514   316     816
```

Now that the items are in a list, we can again use `bind_rows()`, which, in addition to individual objects, can take a single list of objects.

```
## bind our list to single data frame
df <- df_list %>%
    bind_rows()

## show
df
```

```
# A tibble: 24 x 5
   school        year  math  read science
   <chr>        <dbl> <dbl> <dbl>   <dbl>
 1 Bend Gate     1980   515   281     808
 2 Bend Gate     1981   503   312     814
 3 Bend Gate     1982   514   316     816
 4 Bend Gate     1983   491   276     793
 5 Bend Gate     1984   502   310     788
 6 Bend Gate     1985   488   280     789
 7 East Heights  1980   501   318     782
 8 East Heights  1981   487   323     813
 9 East Heights  1982   496   294     818
10 East Heights  1983   497   306     795
# … with 14 more rows
```

Aside from being more aesthetically pleasing, this code is better because it:

- doesn't rely on repeated lines of code with small changes
- is flexible

Imagine you receive three more years of data for each school: 1986, 1987, 1988. As long as the files are in the same format, all you need to do is put the same directory as the other files and rerun your code. It will add them to the `files` vector and your loop will run just as before.

Read in only some files

What if we only want files for Spottsville? Overall, the code is exactly the same except that we want our list to only contain the `spottsville_*.csv` files. How can we do that? With regular expressions!

Notice that the second argument in `list.files()` is `pattern` with a default value of `NULL`. This means it doesn't do any filtering if we leave it out. But if we want to filter which files are stored in `files`, we should use it.

```
## filter files to be read in using pattern
files_sp <- list.files(bys_dir, pattern = "spottsville", full.names = TRUE)

## check
files_sp
```

```
[1] "../data/sch_test/by_school/spottsville_1980.csv"
[2] "../data/sch_test/by_school/spottsville_1981.csv"
[3] "../data/sch_test/by_school/spottsville_1982.csv"
[4] "../data/sch_test/by_school/spottsville_1983.csv"
[5] "../data/sch_test/by_school/spottsville_1984.csv"
[6] "../data/sch_test/by_school/spottsville_1985.csv"
```

Luckily, our regular expression pattern can simply be `"spottsville"`. Were our files less consistently named, we might have had trouble (name those files well!).

The rest of the code should be as it was before (with the small addition of `_sp` in various names to keep distinct from our first attempt).

```
## init list
df_sp_list <- list()

## use loop to read in files
for (i in 1:length(files_sp)) {
    ## read in file (f) and store in list
    df_sp_list[[i]] <- read_csv(files_sp[i])
}

## bind our list to single data frame
df_sp <- df_sp_list %>%
    bind_rows()

## show
df_sp
```

```
# A tibble: 6 x 5
  school       year  math  read science
  <chr>       <dbl> <dbl> <dbl>   <dbl>
1 Spottsville  1980   498   288     813
2 Spottsville  1981   494   270     765
3 Spottsville  1982   507   289     801
4 Spottsville  1983   515   288     775
5 Spottsville  1984   475   289     779
6 Spottsville  1985   515   285     784
```

Use `purrr::map()`

Finally, let's look at the fully {tidyverse} way of solving this problem. We'll use the {purrr} library, which is loaded when we load {tidyverse}, and approaches functional programming more like other languages.

Specifically, we'll use `map()`. According the reference site:

> The `map(.x, .f)` functions transforms each element of the vector `.x` with the function `.f`, returning a vector defined by the suffix (`_lgl`,`_chr()` etc).

Okay…

Restated, `map()` works this way: for each item in an object (`.x`), do something (`.f`) — the *something* being a function. From our file reading loops above, we already have our object and our main thing we want to do (function):

- **object** := file list (`files`)
- **function** := `read_csv()`

What's nice about {purrr} functions is that they work directly with other {tidyverse} functions via pipes, `%>%`. This bit of code reproduces what the first loop did to read in and bind all files:

```
## use purrr::map() to read in all files; then pipe into bind rows
df <- map(files,
          ~ read_csv(.x)) %>%
    bind_rows()

## show
df
```

```
# A tibble: 24 x 5
   school        year  math  read science
   <chr>        <dbl> <dbl> <dbl>   <dbl>
 1 Bend Gate     1980   515   281     808
 2 Bend Gate     1981   503   312     814
 3 Bend Gate     1982   514   316     816
 4 Bend Gate     1983   491   276     793
 5 Bend Gate     1984   502   310     788
 6 Bend Gate     1985   488   280     789
 7 East Heights  1980   501   318     782
 8 East Heights  1981   487   323     813
 9 East Heights  1982   496   294     818
10 East Heights  1983   497   306     795
# … with 14 more rows
```

What's happening here? One at a time;

1. every item in `files` becomes `.x`
2. `read_csv(.x)` is run (in `map()`, the tilde, `~`, tells R that a function is coming next). Since the items in `files` are paths, `read_csv()` works as it always does.

The output of `map()` is a list. Before, we had to initialize a blank list, `df_list()`. But this time we can just use a `%>%` to send the output of `map()` (a list), to `bind_rows()` and assign the **final** result to `df`: our full data frame!

This code is much more compact than our loops above. That said, it also is more abstract (read: less clear). In certain situations, using `map()` in place of a `for()` loop may be faster. But in general, you are just fine using loops. That said, functional programming via the {purrr} suite of functions can be very powerful and may be worth learning more about.