Iteration Presentation

Matthew Malcher

DHSC

2019/09/19 (updated: 2019-09-20)

Function Recap

Recap - tibble

Last week Katie took you through functions in R using the example of a function to normalise a dataframe (or tibble). You made a tibble with 4 columns (a-d), where each one was some random numbers:

```r
#create test dataframe
df <- tibble::tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)
```

```r
dplyr::glimpse(df)
```

```
## Observations: 10
## Variables: 4
## $ a <dbl> 0.9300407, 0.3024134, -0.3435463, -0.1197889, 0.7700452, 1.6...
## $ b <dbl> 0.12283355, 0.04656974, 1.72627270, -0.02851492, 0.61128477,...
## $ c <dbl> 0.39034133, -0.72539581, -1.22859728, -0.33715226, 0.1318634...
## $ d <dbl> 0.46774317, 1.85721425, -1.06434165, 0.37292142, 0.05173845,...
```

Recap - function

You then made a function which could normalise a column:

```r
# create the function rescale01 which
# scales a vector to lie between 0 and 1
rescale01 <- function(x){
  rng <- range(x, na.rm = TRUE) # get min and max of col
  (x - rng[1]) / (rng[2] - rng[1]) # scale column
}
```

And talked about how you could apply it, one column at a time

```r
df$a <- rescale01(df$a)
```

Basic Iteration

For Loops

This time we are going to talk about looping your functions. The basic recipe for a for loop is:

```
for (variable in sequence) { do_thing }
```

For example:

```
for (i in 1:3) { print(i) }
```

```
## [1] 1
## [1] 2
## [1] 3
```

In our example:

- iteration variable: `i`
- sequence: `1:3` (could also generate a sequence using seq)
- thing: `print` the value of `i`

Ways to Loop - Indicies

There are a few different ways you can loop:

We started by looping over *indicies*, generating a sequence of numbers using the `:` operator.

```r
for (i in 1:3) { print(i) }
```

```
## [1] 1
## [1] 2
## [1] 3
```

Ways to Loop - Elements

We could also loop over the *elements* of a list, generated using `list()` or a vector using `c()`

```
example_vector <- c('a', 'b', 'c')
```

```
for (i in example_vector) { print(i) }
```

```
## [1] "a"
## [1] "b"
## [1] "c"
```

This is just a clearer and shorter way of achieving the same thing as:

```
for (i in seq_along(example_vector)) { print(example_vector[[i]]) }
```

```
## [1] "a"
## [1] "b"
## [1] "c"
```

Note - seq_along

Note - `seq_along` just gives a list of indicies to use.

```
seq_along(example_vector)
```

```
## [1] 1 2 3
```

Its a safer version of `1:length(example_vector)`. You should use it instead.

```
1:length(example_vector)
```

```
## [1] 1 2 3
```

```
empty_vec <- c()

for (i in seq_along(empty_vec) ) {print('foo')}
```

```
for (i in 1:length(empty_vec)) {print('bar')}
```

```
## [1] "bar"
## [1] "bar"
```

Ways to Loop - Names

Another variation on the theme is iterating on the *names* of an object. If we create a named vector:

```
a_named_vec <- c('a' = 3, 'b' = 2, 'c' = 1)
```

Then names can be used to index into it:

```
a_named_vec[['b']]
```

```
## [1] 2
```

So we can loop using the *name*:

```
for (i in names(a_named_vec)) {
  print(paste(i, ' = ' , a_named_vec[[i]]))
  }
```

```
## [1] "a  =  3"
## [1] "b  =  2"
## [1] "c  =  1"
```

Looping a Function

Applying our function using a loop

So, we have our rescale function we made last time

```
rescale01
```

```
## function(x){
##   rng <- range(x, na.rm = TRUE) # get min and max of col
##   (x - rng[1]) / (rng[2] - rng[1]) # scale column
## }
```

And a dataframe to apply it to

```
head(df,3)
```

```
## # A tibble: 3 x 4
##        a      b      c      d
##    <dbl>  <dbl>  <dbl>  <dbl>
## 1  0.930 0.123   0.390  0.468
## 2  0.302 0.0466 -0.725  1.86
## 3 -0.344 1.73   -1.23  -1.06
```

Applying our function using a loop

To apply our function to all 4 columns we *could* do:

```r
df[[1]] <- rescale01(df[[1]])
df[[2]] <- rescale01(df[[2]])
df[[3]] <- rescale01(df[[3]])
df[[4]] <- rescale01(df[[4]])
```

Better - a simple loop:

```r
for (i in seq_along(df)) {
  df[[i]] <- rescale01(df[[i]])
}
```

We could also have used the names for indexing in our loop

```r
for (i in names(df)) {
  df[[i]] <- rescale01(df[[i]])
}
```

Less Code - Fewer Mistakes

As you can see, already, even when we are replacing just 4 operations, the loop is less code.

It is also able to handle a dataframe with an arbitrary number of columns.

Finally, its much less likely to contain a copy & paste error. Its easy to make a mistake like:

```
df[[1]] <- rescale01(df[[1]])
df[[2]] <- rescale01(df[[2]])
df[[3]] <- rescale01(df[[2]])
df[[4]] <- rescale01(df[[4]])
```

The `apply` Functions

The `apply` functions

Writing a loop to iterate over and operate on each element of a vector or list is a common task. R has some built in functions to help with that so you dont have to write loads of loops.

The key ones (`??apply` shows them all) are:

- `apply` - Apply Functions Over Array Margins (i.e. over rows or columns)
- `lapply` - Apply a Function over a List or Vector (returns a list)
- `sapply` - A wrapper for lapply which simplifies the output

Our toy examples are all *vectors*, and `df` is a *dataframe* which, can be thought of as a list of columns, (which can also be lists).

As such, we can use `lapply` to do all of the examples we have covered so far.

`lapply` syntax

The syntax for lapply is:

```
lapply(X, FUN, ...)
```

- X - input list or vector - other inputs are coerced to a list using `as.list()`
- FUN - a function we want to apply
- ... - other inputs to FUN

lapply example

So, instead of writing a loop, to print each item of our example vector we can do:

```
lapply(example_vector, print) -> output_list
```

```
## [1] "a"
## [1] "b"
## [1] "c"
```

In this case, we get two outputs, print produces a side effect (printing to the console) as well as a direct output - a list, containing the printed contents of each item in example vector.

So that we see only one of these (the printing to console), I have sent the output of `lapply` to 'output_list'. Otherwise R would print it out.

`lapply` on columns

We can also use lapply to apply each of the columns in df:

```
output_of_lapply <- lapply(df, rescale01)
```

Note - this has returned a list, rather than a dataframe, but we can use `as.data.frame()` to convert it back:

```
df_from_list <- as.data.frame(output_of_lapply)
```

```
head(df_from_list)
```

```
##           a         b         c         d
## 1 0.8157504 0.4180346 0.4180346 0.7379385
## 2 0.6511185 0.3903548 0.3903548 1.0000000
## 3 0.4816778 1.0000000 1.0000000 0.4489794
## 4 0.5403713 0.3631030 0.3631030 0.7200547
## 5 0.7737823 0.5953171 0.5953171 0.6594779
## 6 1.0000000 0.0000000 0.0000000 0.4787566
```

`apply` example

We could also use apply, telling it to work on the 2nd margin (columns), to achieve the same thing:

```
output_of_apply <- apply(df, 2, rescale01)
```

This has returned a matrix so we convert it back to a dataframe:

```
df_from_matrix <- as.data.frame(output_of_apply)
```

```
head(df_from_matrix)
```

```
##           a         b         c         d
## 1 0.8157504 0.4180346 0.4180346 0.7379385
## 2 0.6511185 0.3903548 0.3903548 1.0000000
## 3 0.4816778 1.0000000 1.0000000 0.4489794
## 4 0.5403713 0.3631030 0.3631030 0.7200547
## 5 0.7737823 0.5953171 0.5953171 0.6594779
## 6 1.0000000 0.0000000 0.0000000 0.4787566
```

tidyverse + `apply` = `purrr`

purrr

`purrr` is a part of the *tidyverse* and provides extensive functionality for iterating.

```
library(purrr)
```

```
## Warning: package 'purrr' was built under R version 3.5.3
```

The most basic purrr function is `map`

```
output_of_map <-
  map(.x = df,
      .f = rescale01)
```

`map` does the same thing as `lapply`: it applies a function to each element of a list.

```
all.equal(output_of_map, output_of_lapply)
```

```
## [1] TRUE
```

map

The advantage of `map()` is all the other versions of it!

```
map(.x, .f, ...)
```

```
# conditional
map_if(.x, .p, .f, ..., .else = NULL)
map_at(.x, .at, .f, ...)
```

```
# atomic vectors - will return a list of a given type or die trying
map_lgl(.x, .f, ...)
map_chr(.x, .f, ...)
map_int(.x, .f, ...)
map_dbl(.x, .f, ...)
```

```
# return a dataframe directly
map_df(.x, .f, ...)

# like map, but returns the input, invisibly,
# useful for side effects like print, or plot.
walk(.x, .f, ...)
```

`map_df`

This means that instead of our loop, or our lapply & conversion we can simply use `map_df()` to return our scaled dataframe:

```
output_of_map_df <- map_df(df, rescale01)
```

The output is exactly the same:

```
all.equal(output_of_map_df, df_from_list)
```

```
## [1] TRUE
```

The Elephant in the Room

The Elephant in the Room

As it turns out, applying a function to multiple columns is something that is required all the time.

This means our example is quite easy to do using regular `dplyr`, without the `map` or `apply` functions:

```
dplyr_output <- dplyr::mutate_all(df, rescale01)
```

Its exactly the same:

```
all.equal(output_of_map_df, dplyr_output)
```

```
## [1] TRUE
```

So what is `purrr` for?

So what is purrr for?

Purr has lots more functionality that we havent talked about yet!

Our example so far has been using functions which have a single input only. But what if we have multiple inputs to our function?

Lets imagine we wanted to generate a list of 4 sets of random numbers, with different parameters.

```r
mean_vals <- c(1,2,4,8)
sd_vals   <- c(2,2,5,5)
```

You can generate distrubutions as we did when we made our test dataframe using `rnorm`:

```r
rnorm(n, # number of samples
      mean = 0,
      sd = 1)
```

The Hard Ways - Do Each Case

So, how could we make our list?

The simplest way would be to do each case yourself:

```r
dist_list_basic <- list() # create empty list

dist_list_basic[[1]] <- rnorm(n = 10, mean = 1, sd = 2)
dist_list_basic[[2]] <- rnorm(n = 10, mean = 2, sd = 2)
dist_list_basic[[3]] <- rnorm(n = 10, mean = 4, sd = 5)
dist_list_basic[[4]] <- rnorm(n = 10, mean = 8, sd = 5)
```

The Hard Ways - Loops

You could also write a loop:

```
dist_list_loop <- list() # create empty list

for (i in seq_along(mean_vals)) {

    dist_list_loop[[i]] <-
      rnorm(
        n = 10,
        mean = mean_vals[[i]],
        sd = sd_vals[[i]]
      )
}
```

Easy with `map2()`

map2() lets us pass in two arguments to the function, and iterate on both at once.

```
dist_list_map <-
  map2(.x = mean_vals,
       .y = sd_vals,
       .f = rnorm,
       n = 10) # arguments after the function are repeated
```

Note that `n = 10` is provided **after** `.f` and is therefore kept constant, while `.x` and `.y` are **before** `.f` because there is a new `.x` and `.y` for each iteration.

This works because `n` is labelled explicitly - and the remaining arguments for `rnorm()` (`mean` and `sd`) are matched by position.

More Explicit `map2()`

If you wanted to be more explicit about it you could use a formula ~ instead:

```
dist_list_map <-
  map2(.x = mean_vals,
       .y = sd_vals,
       .f = ~rnorm( n = 10,
                    mean = .x,
                    sd = .y)
  )
```

If you wanted to write the shortest possible thing you could do it on one line:

```
dist_list_map <- map2(mean_vals, sd_vals, rnorm, n = 10)
```

Please dont.

Modelling Example

Modelling example - Model

Working with lists using map, lets you do some really neat stuff.

In this example, we:

- Take the mtcars dataset,
- Split it into a list of three dataframes
  (depending on if there are 4, 6 or 8 cylinders)
- Fit a model to each of the dataframes in the list using map
- …producing a list of model objects

```
by_cylinder <-  split(mtcars, mtcars$cyl)
```

```
models_list <-  map(.x = by_cylinder,
               .f = ~lm(mpg ~ wt, data = .x))
```

Modelling Example - Predict

You can then use map2 to call `predict` to take each of our models, and each bit of the split data, and make predictions.

```
predictions <-
  map2( .x = models_list,      # a model
        .y = by_cylinder,      # corresponding bit of data frame
        .f = predict)          # function to apply
```

The first element of our predictions list contains the mpg predictions for the 4 cylinder cars:

```
predictions[[1]]
```

```
##      Datsun 710       Merc 240D        Merc 230        Fiat 128     Honda Civic
##        26.47010        21.55719        21.78307        27.14774        30.45125
## Toyota Corolla   Toyota Corona       Fiat X1-9   Porsche 914-2    Lotus Europa
##        29.20890        25.65128        28.64420        27.48656        31.02725
##      Volvo 142E
##        23.87247
```

`pmap()`

So thats pretty neat, but what if you wanted to work with more than 2 inputs?

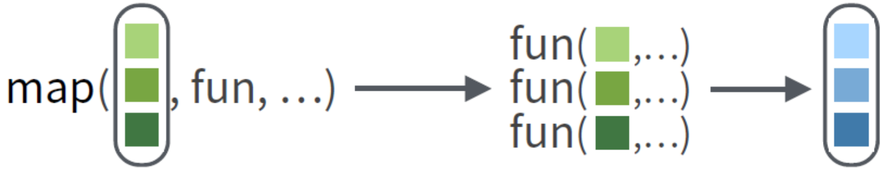
(Previous example used `map`, then `map2`) `purrr` lets you do that using `pmap`. With `pmap` you can provide a **list** of input vectors (or lists) to map a function over.

We have a name for a lists of associated input lists/vectors - a dataframe!

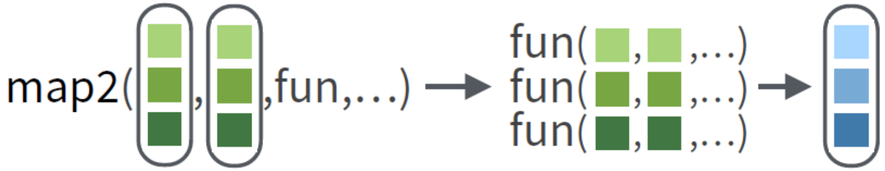
You can pass pmap a dataframe of inputs and it can apply a function to it.

`pmap` in this case is taking each row, and using the values each columns as arguments to your function.

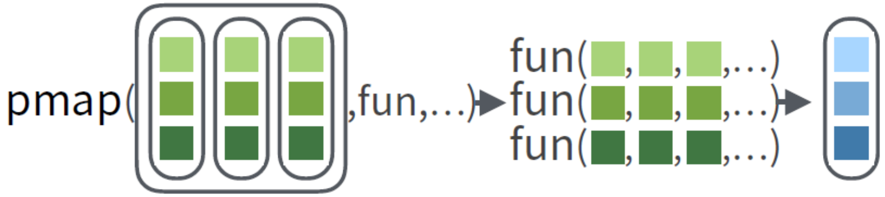
purrr Cheatsheet



map(.x, .f, …) Apply a function to each element of a list or vector. *map(x, is.logical)*



map2(.x, ,y, .f, …) Apply a function to pairs of elements from two lists, vectors. *map2(x, y, sum)*



pmap(.l, .f, …) Apply a function to groups of elements from list of lists, vectors. *pmap(list(x, y, z), sum, na.rm = TRUE)*

Its All Loops

Under the hood, when you use these purrr functions, it is generating and running for loops, so map etc wont make things magically faster than a nested loop.

What purrr *does* bring to the table is the fact that its easer to read, understand and change your code.

Mapping Safely

Mapping Safely!

When you use the map functions to repeat many operations, the chances are much higher that one of those operations will fail.

When this happens, you’ll get an error message, and no output.

So - you need a way to handle the failures.

Fortunately purrr has a way of doing this.

The Problem

In this example we have a list of what we *think* are all numbers,

```
input_list <- list(10, 11, 12, 10, '14')
```

Lets imagine we wanted to multiply them all by 10, and the best way to do this was to write a function to do it:

```
multiply_10 <- function(number) { number * 10 }
```

We could then use `map` to apply our function to our input:

```
map(input_list, multiply_10)
```

Unfortunately, when we get to '14' we get:

```
Error in number * 10 : non-numeric argument to binary operator
```

This means we dont get any of our results.

The Solution

We can use `safely()` to provide some error handling by creating a new function, which always returns a *list*:

```r
safe_multiply <- safely(multiply_10)
```

If our function works the list contains the result and a `NULL` error,

```r
safe_multiply(10)
```

```
## $result
## [1] 100
##
## $error
## NULL
```

The Solution

If it breaks, the list contains a NULL result and the error text.

```
safe_multiply('14')
```

```
## $result
## NULL
##
## $error
## <simpleError in number * 10: non-numeric argument to binary operator>
```

Crucially, this allows R to keep plugging away at the rest of our list, rather than coming to a halt.

So, our new function is *safe* to use with `map()` because it wont bring everything down if it comes across a problem.

Instead `map()` can just rise above it and move on.

Getting at Your Results

This now works:

```
map_output <- map(input_list, safe_multiply)
```

So far so good, but how do you get at your results? They are in a format which isnt super convenient.

```
map_output[[1]]
```

```
## $result
## [1] 100
##
## $error
## NULL
```

transpose

purrr provides the transpose function which restructures our list so that we can get all of our results or errors easily:

```
results <- transpose(map_output)[['result']]
errors  <- transpose(map_output)[['error']]
```

```
results
```

```
## [[1]]
## [1] 100
##
## [[2]]
## [1] 110
##
## [[3]]
## [1] 120
##
## [[4]]
## [1] 100
##
```

```
errors
```

```
## [[1]]
## NULL
##
## [[2]]
## NULL
##
## [[3]]
## NULL
##
## [[4]]
## NULL
##
```

compact

These transposed lists still contain the NULL results, we can filter these out using `compact()` to get simple lists of the result and errors only.

```
results <- compact(results)
errors <- compact(errors)
```

```
results
```

```
## [[1]]
## [1] 100
##
## [[2]]
## [1] 110
##
## [[3]]
## [1] 120
##
## [[4]]
## [1] 100
```

```
errors
```

```
## [[1]]
## <simpleError in number * 10: non-nume
```

Other Uses

This is probably overkill for this example, but `safely` & `map` can be applied to all sorts of things.

For example, you might use them to read a folder full of spreadsheets, returning the errors rather than breaking your script when it finds one it cant read.