Iterators

Hamid Semiyari

Learning Objectives

- Learn about iteration.
- Iterators in base R.
- Iterators in purrr.
- Chapter 21 of RDS.
- Purrr Cheat Sheet.
- Purrr Overview.

For Loops

- Load the tidyverse

```
library(tidyverse)
```

- *Iteration* is the repetition of some amount of code.
- If we didn't know the `sum()` function, how would we add up the elements of a vector?

```
x <- c(8, 1, 3, 1, 3)
```

- We could manually add the elements.

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

```
[1] 16
```

But this is prone to error (through copy and paste). Also, what if `x` has 10,000 elements?

- For loops to the rescue!

```
sumval <- 0
for (i in seq_along(x)) {

   sumval <- sumval + x[[i]]
   print(paste("x[",i,"]=",x[i],"and x[[",i,"]]=", x[[i]], "The sum is", sumval)) # wa
}
```

```
[1] "x[ 1 ]= 8 and x[[ 1 ]]= 8 The sum is 8"
[1] "x[ 2 ]= 1 and x[[ 2 ]]= 1 The sum is 9"
[1] "x[ 3 ]= 3 and x[[ 3 ]]= 3 The sum is 12"
[1] "x[ 4 ]= 1 and x[[ 4 ]]= 1 The sum is 13"
[1] "x[ 5 ]= 3 and x[[ 5 ]]= 3 The sum is 16"
```

```
sumval
```

```
[1] 16
```

- Each for loop contains the following elements:

  1. **Output**: This is `sumval` above. We allocate the space for the output *before* the for loop.
  2. **Sequence**: This is `seq_along(x)` above, which evaluates to `1 2 3 4 5`. These are the values that `i` will go through each iteration.
  3. **Body**: This is the code between the curly braces `{}`. This is the code that will be evaluated each iteration with a new value of `i`.

- In the above sequence, R internally transforms the code to:

```
sumval <- 0
sumval <- sumval + x[[1]]
sumval <- sumval + x[[2]]
sumval <- sumval + x[[3]]
sumval <- sumval + x[[4]]
sumval <- sumval + x[[5]]
sumval
```

```
[1] 16
```

- You often want to fill a vector with values. You should create this vector beforehand using the `vector()` function.
- For example, let's calculate a vector of cumulative sums of `x`.

```
cumvec <- vector(mode = "double", length = length(x))
cumvec
```

```
[1] 0 0 0 0 0
```

```
x
```

```
[1] 8 1 3 1 3
```

```
for (i in seq_along(cumvec)) {
   if (i == 1) {
     cumvec[[i]] <- x[[i]]
   } else {
     cumvec[[i]] <- cumvec[[i - 1]] + x[[i]]
   }
}
cumvec
```

```
[1]  8  9 12 13 16
```

```
## Same as cumsum(x)
cumsum(x)
```

```
[1]  8  9 12 13 16
```

- **Exercise**: The first two numbers of the Fibonacci Sequence are 0 and 1. Each succeeding number is the sum of the previous two numbers in the sequence. For example, the third element is $1 = 0 + 1$, while the fourth elements is $2 = 1 + 1$, and the fifth element is $3 = 2 + 1$. Use a for loop to calculate the first 100 Fibonacci Numbers. Sanity Check: The $\log_2$ of the 100th Fibonacci Number is about 67.57.
- Looping is often done over the columns of a data frame.
- Note: for a data frame `df`, `seq_along(df)` is the same as `1:ncol(df)` which is the same as `1:length(df)` (since data frames are special cases of lists).
- Let's calculate the mean of each column of `mtcars`

```
data("mtcars")
mean_vec <- vector(mode = "numeric", length = length(mtcars))
for (i in seq_along(mtcars)) {
  mean_vec[[i]] <- mean(mtcars[[i]], na.rm = TRUE)
}
mean_vec
```

```
 [1]  20.090625   6.187500 230.721875 146.687500   3.596563   3.217250
 [7]  17.848750   0.437500   0.406250   3.687500   2.812500
```

```
colMeans(mtcars)
```

```
      mpg       cyl      disp        hp      drat        wt      qsec
20.090625  6.187500 230.721875 146.687500  3.596563  3.217250 17.848750
       vs        am      gear      carb
 0.437500  0.406250  3.687500  2.812500
```

- Why not just use `colMeans()`? Well, there is no "`colSDs`" function, so iteration is important for applying non-implemented functions to multiple elements in R.
- **Exercise**: Use a for loop to calculate the standard deviation of each penguin trait in the `penguins` data frame from the `palmerpenguins` package.

purrr

Basic Mappings

- R is a functional programming language. Which means that you can pass functions to functions.
- Suppose on `mtcars` we want to calculate the column-wise mean, the column-wise median, the column-wise standard deviation, the column-wise maximum, the column-wise minimum, and the column-wise MAD. The for-loop would look very similar

```
funvec <- rep(NA, length = length(mtcars))
for (i in seq_along(funvec)) {
  funvec[i] <- fun(mtcars[[i]], na.rm = TRUE)
}
funvec
```

- Ideally, we would like to just tell R what function to apply to each column of `mtcars`. This is what the purrr package allows us to do.
- purrr is a part of the tidyverse, and so does not need to be loaded separately.
- `map_*()` takes a vector (or list or data frame) as input, applies a provided function on each element of that vector, and outputs a vector of the same length.
  - `map()` returns a list.
  - `map_lgl()` returns a logical vector.
  - `map_int()` returns an integer vector.
  - `map_dbl()` returns a double vector.
  - `map_chr()` returns a character vector.

```
map_dbl(mtcars, mean)
```

```
map_dbl(mtcars, median)
```

```
map_dbl(mtcars, sd)
```

```
map_dbl(mtcars, mad)
```

```
map_dbl(mtcars, min)
```

```
map_dbl(mtcars, max)
```

- You can pass on more arguments in `map_*()`.

```
map_dbl(mtcars, mean, na.rm = TRUE)
```

- Suppose you want to get the output of `summary()` on each column.

```
map(mtcars, summary)
```

- **Exercise** (RDS 21.5.3.1): Write code that uses one of the map functions to:
  1. Determine the type of each column in `nycflights13::flights`.
  2. Compute the number of unique values in each column of `palmerpenguins::penguins`.
  3. Generate 10 random normals for each of $\mu = -10, 0, 10, \ldots, 100$.

Shortcuts

- You can refer to elements of the vector by "`.`" in a `map()` call if the `.f` argument is preceded by a "`~`". For example, the following are three equivalent ways to calculate the mean of each column in `mtcars`.

```
map_dbl(mtcars, mean)
map_dbl(mtcars, function(.) mean(.))
map_dbl(mtcars, ~mean(.))
```

- What is actually going on is that purrr is creating an "anonymous function"

```
.f <- function(.) {
  mean(.)
}
```

and then calling this function in `map()`.

```
map_dbl(mtcars, .f)
```

- Why is this useful? Consider the following chunk of code which allows us to fit many simple linear regression models:

```
mtcars %>%
  split(.$cyl) %>%
  map(function(df) lm(mpg ~ wt, data = df)) ->
  lmlist
```

- `split(.$cyl)` will turn the data frame into a list of data frames where each data frame has a different value of `cyl` for all units. The "`.`" references the current data frame.
- `function(df) lm(mpg ~ wt, data = df)` defines a function (called an "anonymous function") that will fit a linear model of `mpg` on `wt` where those variables are in the data frame `df`.
- The `map()` call fits that linear model to each of the three data frames in the list created by `split()`.
- What is returned is a list of three `lm` objects that you can use to get fits and summaries.

```
summary(lmlist[[1]])
```

```
Call:
lm(formula = mpg ~ wt, data = df)

Residuals:
    Min      1Q  Median      3Q     Max
-4.1513 -1.9795 -0.6272  1.9299  5.2523

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   39.571      4.347   9.104 7.77e-06 ***
wt            -5.647      1.850  -3.052   0.0137 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 3.332 on 9 degrees of freedom
Multiple R-squared:  0.5086,    Adjusted R-squared:  0.454
F-statistic: 9.316 on 1 and 9 DF,  p-value: 0.01374
```

- Again, rather than create an "anonymous function", you can use the formula notation to do the same thing:

```
mtcars %>%
  split(.$cyl) %>%
  map(~lm(mpg ~ wt, data = .)) ->
  lmlist
```

- Here, the "`.`" in "`data = .`" references the current data frame from the list of data frames that we are iterating through.
- We can use `map()` to get a list of summaries.

```
lmlist %>%
  map(summary) ->
  sumlist
```

- If you want to extract the R^2, you can do this using the formula notation as well.

```
sumlist[[1]]$r.squared ## only gets one R^2 out.
```

```
[1] 0.5086326
```

```
    ## Gets all R^2 out
    sumlist %>%
      map(~.$r.squared)
```

```
$`4`
[1] 0.5086326

$`6`
[1] 0.4645102

$`8`
[1] 0.4229655
```

- **Exercise**: A *t*-test is used to test for differences in population means. R implements this with `t.test()`. For example, if I want to test for differences between the mean `mpg`'s of automatics and manuals (coded in variable `am`), I would use the following syntax.

```
  t.test(mpg ~ am, data = mtcars)$p.value
```

Use `map()` to get the p-value for this test within each group of `cyl`.

`keep()` and `discard()`.

- `keep()` selects all variables that return `TRUE` according to some function.
- E.g. let's keep all numeric variables and calculate their means in the `palmerpenguins::penguins` data frame.

```
  library(palmerpenguins)
  data("penguins")
  penguins %>%
    keep(is.numeric) %>%
    map_dbl(mean, na.rm = TRUE)
```

```
bill_length_mm    bill_depth_mm flipper_length_mm       body_mass_g
      43.92193          17.15117         200.91520        4201.75439
          year
   2008.02907
```

- `discard()` will select all variables that return `FALSE` according to some function.
- Let's count the number of each value for each categorical variable:

```
    penguins %>%
      discard(is.numeric) %>%
      map(table)
```

```
$species

   Adelie Chinstrap    Gentoo
      152        68       124

$island

   Biscoe     Dream Torgersen
      168       124        52

$sex

female   male
   165    168
```

- Other less useful functions are available in Section 21.9 of RDS.
- **Exercise**: In the `mtcars` data frame, keep only variables that have a mean greater than `10` and calculate their mean. Hint: You'll have to use some of the shortcuts above.