# Iterators

*David Gerard*

*2019-03-07*

## Learning Objectives

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

## 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]]
  }
  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]]
  ```

```r
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`.

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

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

```r
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
```

```r
## 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 most 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`

```r
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]  20.0906   6.1875 230.7219 146.6875   3.5966   3.2172  17.8487
##  [8]   0.4375   0.4062   3.6875   2.8125
```

```r
colMeans(mtcars)
```

```
##      mpg      cyl     disp       hp     drat       wt     qsec       vs
##  20.0906   6.1875 230.7219 146.6875   3.5966   3.2172  17.8487   0.4375
##       am     gear     carb
```

```
## 0.4062 3.6875 2.8125
```

- 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 plant trait in the `iris` data frame.

# 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 arguements 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 `iris`.
  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`.

```r
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"

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

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

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

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

```r
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.

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

```
##
## Call:
## lm(formula = mpg ~ wt, data = df)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -4.151 -1.980 -0.627  1.930  5.252
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)    39.57       4.35    9.10  7.8e-06
## wt             -5.65       1.85   -3.05    0.014
##
## Residual standard error: 3.33 on 9 degrees of freedom
## Multiple R-squared:  0.509,  Adjusted R-squared:  0.454
## F-statistic: 9.32 on 1 and 9 DF,  p-value: 0.0137
```

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

```r
mtcars %>%
  split(.$cyl) %>%
```

```r
  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.

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

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

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

```
## [1] 0.5086
```

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

```
## $`4`
## [1] 0.5086
##
## $`6`
## [1] 0.4645
##
## $`8`
## [1] 0.423
```

- **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 follwoing syntax.

```r
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 `iris` data frame.

```r
iris %>%
  keep(is.numeric) %>%
  map_dbl(mean)
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##        5.843        3.057        3.758        1.199
```

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

```r
iris %>%
  discard(is.numeric) %>%
  map(table)
```

```
## $Species
##
##     setosa versicolor  virginica
##         50         50         50
```

- 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.