Control Flow: Vectorized Functions

Vectorized Functions

In the spirit of loops, vectorized functions give us a way to execute code on an entire 'vector' at once (although we can be a bit more general than just vectors). This tends to speed up computation in comparison to basic loops in R!

This is because loops are inefficient in R. R is an interpreted language. This means that it does a lot of the work of figuring out what to do for you. (Think about function dispatch - it looks at the type of object and figures out which version of `plot()` or `summary()` to use.) This process tends to slow R down in comparison to a vectorized operation where it still runs a loop under the hood but a vector should have all the same type of elements in it. This means it can avoid figuring the same thing out repeatedly!

Vectorized Functions for Common Numeric Summaries

There are some 'built-in' vectorized functions that are quite useful to apply to a 2D type object:

- `colMeans()`, `rowMeans()`
- `colSums()`, `rowSums()`
- `colSds()`, `colVars()`, `colMedians()` (must install the `matrixStats` package to get these)

Let's go back to our batting dataset from the previous note set.

```
library(Lahman)
```

```
Warning: package 'Lahman' was built under R version 4.1.3
```

```
my_batting <- Batting[, c("playerID", "teamID", "G", "AB", "R", "H", "X2B",
head(my_batting)
```

```
   playerID teamID  G  AB  R  H X2B X3B HR
1 abercda01    TRO  1   4  0  0   0   0  0
2  addybo01    RC1 25 118 30 32   6   0  0
3 allisar01    CL1 29 137 28 40   4   5  0
4 allisdo01    WS3 27 133 28 44  10   2  2
5 ansonca01    RC1 25 120 29 39  11   3  0
6 armstbo01    FW1 12  49  9 11   2   1  0
```

We can apply the `colMeans()` function easily!

```
colMeans(my_batting[, 3:9])
```

```
         G         AB          R          H        X2B        X3B
HR
 50.740488 130.241220  18.483406  36.288605   6.202024   1.247075
```

```
 50.740488 139.241320  18.483496  36.388605   6.202024   1.247075
2.850150
```

If we **install** the `matrixStats` package (download the files from the internet), we can then use the `colMedians()` function to obtain the column medians in a quick fashion.

```
#install.packages("matrixStats") #only run this once on your machine!
library(matrixStats)
```

```
Warning: package 'matrixStats' was built under R version 4.1.3
```

```
colMedians(my_batting[, 3:9])
```

```
Error in colMedians(my_batting[, 3:9]): Argument 'x' must be a matrix or a
vector.
```

Ah, this package requires the object passed to be a matrix or vector (homogenous). Although our data frame we pass is homogenous, the function doesn't have a check for that. No worries, we can convert to a matrix using `as.matrix()` (similar to the `is.` family of functions there is an `as.` family of functions (read as 'as dot')).

```
colMedians(as.matrix(my_batting[, 3:9]))
```

```
[1] 34 46  4  8  1  0  0
```

Let's compare the speed of this code to the speed of a for loop!

- The `microbenchmark` package allows for easy recording of computing time.
- We just wrap the code we want to benchmark in the `microbenchmark()` function.
- Here we will grab all the numeric columns from the data
- Some columns contain `NA` or missing values. We'll add `na.rm = TRUE` to both function calls to ignore those values (this is where the for loop actually struggles in this case!)

```
#install.packages("microbenchmark") #run only once on your machine!
library(microbenchmark)
my_numeric_batting <- Batting[, 6:22] #get all numeric columns
vectorized_results <- microbenchmark(
  colMeans(my_numeric_batting, na.rm = TRUE)
)

loop_results <- microbenchmark(
  for(i in 1:17){
    mean(my_numeric_batting[, i], na.rm = TRUE)
  }
)
```

- Compare computational time

```
vectorized_results
```

```
Unit: milliseconds
                                     expr    min      lq     mean median
 colMeans(my_numeric_batting, na.rm = TRUE) 3.4724 3.63205 4.530967 3.7457
     uq     max neval
 4.4266 10.0154   100
```

```
loop_results
```

```
Unit: milliseconds
                                                                  expr
min
 for (i in 1:17) {     mean(my_numeric_batting[, i], na.rm = TRUE) }
11.0554
       lq     mean   median       uq     max neval
 15.60525 16.50845 16.20455 17.1105 45.3125   100
```

Vectorized `ifelse`

We saw the limitation of using standard `if/then/else` logic for manipulating a data set. The `ifelse()` function is a vectorized form of if/then/else logic.

Let's revisit our example that used the `airquality` dataset. We wanted to code a wind category variable:

- high wind days (15mph $\leq$ wind)
- windy days (10mph $\leq$ wind < 15mph)
- lightwind days (6mph $\leq$ wind < 10mph)
- calm days (wind $\leq$ 6mph)

The syntax for `ifelse` is:

```
ifelse(vector_condition, if_true_do_this, if_false_do_this)
```

a vector is returned!

```
ifelse(airquality$Wind >= 15, "HighWind",
          ifelse(airquality$Wind >= 10, "Windy",
                 ifelse(airquality$Wind >= 6, "LightWind",
                        ifelse(airquality$Wind >= 0, "Calm", "Error"))))
```

```
  [1] "LightWind" "LightWind" "Windy"     "Windy"     "Windy"     "Windy"
  [7] "LightWind" "Windy"     "HighWind"  "LightWind" "LightWind"
"LightWind"
 [13] "LightWind" "Windy"     "Windy"     "Windy"     "Windy"
"HighWind"
 [19] "Windy"     "LightWind" "LightWind" "HighWind"  "LightWind" "Windy"
 [25] "HighWind"  "Windy"     "LightWind" "Windy"     "Windy"     "Calm"
 [31] "LightWind" "LightWind" "LightWind" "HighWind"  "LightWind"
"LightWind"
 [37] "Windy"     "LightWind" "LightWind" "Windy"     "Windy"     "Windy"
 [43] "LightWind" "LightWind" "Windy"     "Windy"     "Windy"
```

```
"HighWind"
 [49] "LightWind" "Windy"     "Windy"     "LightWind" "Calm"      "Calm"
 [55] "LightWind" "LightWind" "LightWind" "Windy"     "Windy"     "Windy"
 [61] "LightWind" "Calm"      "LightWind" "LightWind" "Windy"     "Calm"
 [67] "Windy"     "Calm"      "LightWind" "Calm"      "LightWind"
"LightWind"
 [73] "Windy"     "Windy"     "Windy"     "Windy"     "LightWind" "Windy"
 [79] "LightWind" "Calm"      "Windy"     "LightWind" "LightWind" "Windy"
 [85] "LightWind" "LightWind" "LightWind" "Windy"     "LightWind"
"LightWind"
 [91] "LightWind" "LightWind" "LightWind" "Windy"     "LightWind"
"LightWind"
 [97] "LightWind" "Calm"      "Calm"      "Windy"     "LightWind"
"LightWind"
[103] "Windy"     "Windy"     "Windy"     "LightWind" "Windy"     "Windy"
[109] "LightWind" "LightWind" "Windy"     "Windy"     "HighWind"  "Windy"
[115] "Windy"     "LightWind" "Calm"      "LightWind" "Calm"
"LightWind"
[121] "Calm"      "LightWind" "LightWind" "LightWind" "Calm"      "Calm"
[127] "Calm"      "LightWind" "HighWind"  "Windy"     "Windy"     "Windy"
[133] "LightWind" "Windy"     "HighWind"  "LightWind" "Windy"     "Windy"
[139] "LightWind" "Windy"     "Windy"     "Windy"     "LightWind" "Windy"
[145] "LightWind" "Windy"     "Windy"     "HighWind"  "LightWind" "Windy"
[151] "Windy"     "LightWind" "Windy"
```

Whoa that was pretty easy! Nice.

Let's compare this to using a for loop speed-wise.

```
loopTime<-microbenchmark(
  for (i in seq_len(nrow(airquality))){
    if(airquality$Wind[i] >= 15){
       "HighWind"
    } else if (airquality$Wind[i] >= 10){
      "Windy"
    } else if (airquality$Wind[i] >= 6){
      "LightWind"
    } else if (airquality$Wind[i] >= 0){
      "Calm"
    } else{
      "Error"
    }
  }
, unit = "us")
```

```
vectorTime <- microbenchmark(
  ifelse(airquality$Wind >= 15, "HighWind",
         ifelse(airquality$Wind >= 10, "Windy",
                ifelse(airquality$Wind >= 6, "LightWind",
                       ifelse(airquality$Wind >= 0, "Calm", "Error"))))
)
```

Compare!

```
loopTime
```

```
Unit: microseconds

expr
 for (i in seq_len(nrow(airquality))) {     if (airquality$Wind[i] >= 15) {
"HighWind"     }     else if (airquality$Wind[i] >= 10) {         "Windy"
}     else if (airquality$Wind[i] >= 6) {         "LightWind"     }
else if (airquality$Wind[i] >= 0) {         "Calm"     }     else {
"Error"     } }
    min     lq     mean median     uq    max neval
 3665.3 4061.5 4428.095 4188.1 4449.6 7261.4   100
```

```
vectorTime
```

```
Unit: microseconds

expr
 ifelse(airquality$Wind >= 15, "HighWind", ifelse(airquality$Wind >=
10, "Windy", ifelse(airquality$Wind >= 6, "LightWind",
ifelse(airquality$Wind >=      0, "Calm", "Error"))))
  min   lq    mean median     uq   max neval
 91.9 93.3 110.636  98.15 111.35 297.4   100
```

Note: There is an `if_else()` function from the `dplry` package. This has more restrictions than `ifelse()` but otherwise is pretty similar.

Recap!

- Loops are slower in R
- Use vectorized functions if possible
- Common vectorized functions
    - `colMeans()`, `rowMeans()`
    - `colSums()`, `rowSums()`
    - `matrixStats::colSds()`, `matrixStats::colVars()`, `matrixStats::colMedians()`
    - `ifelse()` or `dplyr::if_else()`
    - `apply` family (covered soon)
    - `purrr` package (covered in a bit)