# Session 6: Looping Structures

*Samuel P Callisto*

*July 19, 2018*

## Contents

# Part 2: Looping Structures

Great reference: 'R for Data Science: Iteration'

Iteration is an important aspect of coding because it allows you to repeat operations multiple times without copy-pasting sections of your code. There are many looping structures available in R, but the most commonly used is the for-loop.

## For-loops

### Three essential components for a for-loop

- Output
- Sequence
- Body

```
## create example dataset
df <- data.frame(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)

output <- vector("double", 4)      # 1. output
for (i in 1:ncol(df)) {            # 2. sequence
  output[i] <- median(df[,i])      # 3. body
}
output
```

```
## [1] -0.11937503 -0.08361914  0.40557951  0.23181049
```

### About the output container

It may be tempting to skip this step and grow the size of your output variable with each iteration. However, this uses much more memory and can cause your computer to crash on very large datasets ("computationally expensive"). Best practice is to always create a container for your output prior to generating it, such as a vector, matrix, list, or data.frame. If you are unable to know the length of your output before starting the loop, save your output in a list data type (see 'R for Data Science: Unknown Output Length').

### Slightly fancier method

- calculating number of required spaces from dataset rather than hard-coding 4; this allows for more flexible input
- seq_along() is a wrapper function for length() which avoids zero-length vector errors
- can use element selection [[]] for both vectors and data.frames rather than using different syntax

```
output <- vector("double", ncol(df))   # 1. output
for (i in seq_along(df)) {             # 2. sequence
  output[[i]] <- median(df[[i]])       # 3. body
}
output
```

```
## [1] -0.11937503 -0.08361914  0.40557951  0.23181049
```

**Exercise 1**

Create a for-loop that iterates through the mtcars dataset and calculates the mean for each column

**Exercise 2**

Generate 10 random normals from each of $\mu$ = -10, 0, 10, 100

### Alternate for-loop structures

The basic for-loop uses the variable i as an index through a vector or data.frame. In some cases it might be advantageous to access values directly rather than by using an index. There are two alternate methods for iteration using for-loops: 1: n in names(xs) 2: x in xs

```
## create empty vector for output with meaningful name
mtcarsMeans <- vector("numeric", ncol(mtcars))

## add column names to output vector
names(mtcarsMeans) <- names(mtcars)

## loop through columns to calculate mean for each
for(i in names(mtcars)){
  mtcarsMeans[[i]] <- mean(mtcars[[i]])
}

## display output
mtcarsMeans
```

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

## Breaking out of loops

### Break

Sometimes there will be a case in which you want to stop (break) or skip (next) when you encounter an element.

```
lettersBeforeP <- ""                                    # output
for(i in LETTERS){                                      # sequence
  lettersBeforeP <- paste(lettersBeforeP,i,sep= " ")   # body
  if(i == "P"){
    break
  }
}
lettersBeforeP
```

```
## [1] " A B C D E F G H I J K L M N O P"
```

### Next

Using break will cause the loop to end, but you can use next to skip to the next iteration and continue looping

```
alphaWithoutSAM <- ""                            # output
for(i in letters){                               # sequence
  if(i == "s" | i == "a" | i == "m") next        # body
  alphaWithoutSAM <- paste(alphaWithoutSAM, i, " ")
}
alphaWithoutSAM
```

```
## [1] " b   c   d   e   f   g   h   i   j   k   l   n   o   p   q   r   t   u   v   w   x   y   z  "
```

## Nesting for-loops

Sometimes you will be utilizing multi-level data that varies by multiple factors. In these cases we can utilize multiple for-loops nested within each other.

### Example: creating a times table from one through ten

```
## create output container
timesTable <- matrix(-99,nrow=10, ncol=10)

## iterate through row dimension
for(i in 1:10){
  ## iterate through column dimension
  for(j in 1:10){
    timesTable[i,j] = i*j
  }
}
timesTable
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,]    1    2    3    4    5    6    7    8    9    10
##  [2,]    2    4    6    8   10   12   14   16   18    20
##  [3,]    3    6    9   12   15   18   21   24   27    30
##  [4,]    4    8   12   16   20   24   28   32   36    40
##  [5,]    5   10   15   20   25   30   35   40   45    50
##  [6,]    6   12   18   24   30   36   42   48   54    60
##  [7,]    7   14   21   28   35   42   49   56   63    70
##  [8,]    8   16   24   32   40   48   56   64   72    80
##  [9,]    9   18   27   36   45   54   63   72   81    90
## [10,]   10   20   30   40   50   60   70   80   90   100
```

### Example: loading multiple files into R

Say we have 8 files with a consistent naming pattern: 01_session_TPM.csv 01_session_PBO.csv 02_session_TPM.csv 02_session_PBO.csv 04_session_TPM.csv 04_session_PBO.csv 07_session_TPM.csv 07_session_PBO.csv We want to import all these files to analyze. Since the files are varying by two dimensions, we can use nested for-loops to generate all possible combinations.

```
## combinations of subject IDs and drugs
SID <- c("01", "02", "04", "07")
```

```r
drug <- c("TPM", "PBO")

## output container
allSubjectData <- list()

## error catching statement
try(
## sequence through each subject ID
for(i in SID){
  ## sequence through each drug
  for(j in drug){
    ## generate a name for each element of the list
    combo <- paste0(i,"-",j)
    ## import files into the named element of the list
    allSubjectData[[combo]] <- read.csv(paste0(i,"_session_",j,".csv"))
  }
}
, silent = T)
```

```
## Warning in file(file, "rt"): cannot open file '01_session_TPM.csv': No such
## file or directory
```

## While-loops

### While

This type of looping structure is useful for situations when the number of iterations necessary for the task to finish is unknown. It should be noted that all for-loops can be re-written as as a while-loop, but the opposite is not true.

```r
countToTen <- ""
i <- 1
while(i <= 10){
  countToTen <- paste(countToTen, i, sep=" ")
  i <- i + 1
}
countToTen
```

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

### Repeat

A modifed version of the while loop is repeat, which will keep running until it reaches a break statement.

```r
countToTen <- ""
i <- 1
repeat{
  countToTen <- paste(countToTen, i, sep=" ")
  if(i==10){
    break
  }else{
    i <- i + 1
  }
}
countToTen
```

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

**Exercise 3**

Calculate the mean for all columns in iris that contain numeric data using a while-loop

## Apply and Purrr

Base R gives us the apply() family of functions, which can be useful alternatives to for-loops. Let's revisit the example from earlier of calculating the mean for all columns in the mtcars dataset

```
apply(mtcars,2,mean)
```

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

If your function can easily adhere to the syntax of the apply() functions, it can give the same results as a for-loop much less code. * apply(): evaluate a function over the margins of a matrix or array * lapply(): evaluate a function on each element in a list, and return the results as a list * sapply(): evaluate a function on each element in a list, and return the results in a "simplified form" (not always predictable, but can be convenient) * vapply(): similar to sapply(), but with more consistent return types * tapply(): evaluate a function on subsets of a vector; alternative to group_by() for dealing with subsets

Another useful package in the tidyverse is 'purrr', which is similar to apply, but strives to be more consistent with argument structure and syntax.

The difference in run-time between all these different methods of iteration is fairly similar (though purrr is likely the fastest), so just choose whichever syntax you prefer.