

# Stat 260, Lecture 12, Iteration

Brad McNeney



# Load packages and datasets

```
library(tidyverse)
```

# Reading

- ▶ Iteration chapter of R for Data Science, chapter 17 of printed book and chapter 21 online.
- ▶ purrr cheatsheet [<https://github.com/rstudio/cheatsheets/raw/master/purrr.pdf>]

## Iterating over a vector

- ▶ For loops allow iteration.
- ▶ A common scenario for iteration is that our data is in a vector (list), and we want to perform the same operation on each element.
- ▶ Such iteration is so common that special tools have been developed with the aim of reducing the amount of code (and therefore errors) required for common iterative tasks.
  - ▶ Tools in base R include the `apply()` family of functions.
  - ▶ A tidyverse package called `purrr` includes more.

# Example data

- To illustrate iteration we simulate data and fit four regression models.

```
set.seed(42)
n <- 100
x1 <- rnorm(n); x2<-rnorm(n)
y1 <- x1 + rnorm(n,sd=.5); y2 <- x1+x2+rnorm(n,sd=.5)
y3 <- x2 + rnorm(n,sd=.5); y4 <- rnorm(n,sd=.5)
rr <- list(fit1 = lm(y1 ~ x1+x2),
  fit2 = lm(y2 ~ x1+x2),
  fit3 = lm(y3 ~ x1+x2),
  fit4 = lm(y4 ~ x1+x2))
coef(rr$fit1)
```

```
## (Intercept)          x1          x2
## 0.0008831357 0.9281453769 0.0426465892
```

## Exercise

- ▶ The elements of the list `rr` from last slide are `lm` objects. The function `coef()` is generic. Assign class `"lm_vec"` to `rr` and write a `coef()` method for objects of this class. Hint: Your function could encapsulate a `for()` loop like the following.

```
for(i in seq_along(rr)) { # safer than 1:length(rr)
  print(coef(rr[[i]]))
}
```

```
## (Intercept)          x1          x2
## 0.0008831357 0.9281453769 0.0426465892
## (Intercept)          x1          x2
## 0.01572372 1.03114836 1.00306653
## (Intercept)          x1          x2
## -0.06641184 0.04316514 0.93035180
## (Intercept)          x1          x2
## -0.008394232 -0.018428268 -0.116309416
```

## Extracting the regression coefficient for x1

- Using a `for()` loop, we initialize an object to hold the **output**, loop along a **sequence** of values for an index variable, and execute the **body** for each value of the index variable.

```
betahat <- vector("double",length(rr))
for(i in seq_along(rr)) { # safer than 1:length(rr)
  betahat[i] <- coef(rr[[i]])["x1"]
}
betahat
```

```
## [1] 0.92814538 1.03114836 0.04316514 -0.01842827
```



## Looping over elements of a set

- ▶ The index set in the `for()` loop can be general.
  - ▶ We might use this generality to loop over named components of a list.

```
fits <- paste0("fit",1:4)
for(ff in fits) {
  print(coef(rr[[ff]])["x1"])
}
```

```
##          x1
## 0.9281454
##          x1
## 1.031148
##          x1
## 0.04316514
##          x1
## -0.01842827
```

- ▶ Looping over a set makes it harder to save the results, though.

# Avoid growing vectors incrementally

```
means <- seq.int(1000)
set.seed(123)
system.time({
  output <- double()
  for (i in seq_along(means)) {
    n <- sample(100, 1)
    output <- c(output, rnorm(n, means[[i]]))
  }
})
```

```
##      user  system elapsed
##    0.153    0.102    0.305
```

```
system.time({  
  out <- vector("list", length(means))  
  for (i in seq_along(means)) {  
    n <- sample(100, 1)  
    out[[i]] <- rnorm(n, means[[i]])  
  }  
  out <- unlist(out)  
})
```

```
##      user  system elapsed  
##    0.024    0.002    0.025
```

## bind\_cols() and bind\_rows()

```
out <- vector("list", length(means))
n <- 100
for (i in seq_along(means)) {
  out[[i]] <- rnorm(n, means[[i]])
}
out <- bind_cols(out)
out <- vector("list", length(means))
for (i in seq_along(means)) {
  out[[i]] <- tibble(y=rnorm(n, means[[i]]),x=rnorm(n))
}
out <- bind_rows(out)
```

## The body of a loop can be a small part of the code

- ▶ In our examples, most of the code is for setting up the output and looping, with very little to do with the body.
- ▶ To illustrate, consider a small change: instead of the estimated coefficient of  $x_1$  we wanted the estimated coefficient of  $x_2$ :

```
betahat <- vector("double",length(rr))  
for(i in seq_along(rr)) { # after than 1:length(rr)  
  betahat[i] <- coef(rr[[i]])["x2"]  
}  
betahat
```

```
## [1] 0.04264659 1.00306653 0.93035180 -0.11630942
```

## Exercise

- ▶ Write a `for()` loop to find the `mode()` of each column in `nycflights13::flights`

## Using lapply()

- ▶ The intent of lapply() is to take care of the output and the loop, allowing us to focus on the body.

```
b1fun <- function(fit) { coef(fit)["x1"] } # body  
bfun <- function(fit,cc) { coef(fit)[cc] } # body  
lapply(rr,b1fun) # or sapply(rr,b1fun) or unlist(lapply(rr,b1fun))
```

```
## $fit1  
##      x1  
## 0.9281454  
##  
## $fit2  
##      x1  
## 1.031148  
##  
## $fit3  
##      x1  
## 0.04316514  
##  
## $fit4  
##      x1  
## -0.01842827
```

```
lapply(rr,bfun,"x1")
```

```
## $fit1
```

## Exercise

- ▶ Re-write your `coef()` method for objects of class `lm_vec` to use `lapply()`.



## Iterating with the `map()` functions from `purrr`

- ▶ The `purrr` package provides a family of functions `map()`, `map_dbl()`, etc. that do the same thing as `lapply()` but work better with other tidyverse functions.
  - ▶ `map()` returns a list, like `lapply()`.
  - ▶ `map_dbl()` returns a double vector, etc.

```
library(purrr)
map_dbl(rr, bfun) # or rr %>% map_dbl(bfun)
```

```
##           fit1           fit2           fit3           fit4
## 0.92814538  1.03114836  0.04316514 -0.01842827
```

```
# map_dbl(rr, bfun, "x1")
```

# Exercises

- ▶ Use `map_chr()` to return the `mode()` of each column of the `nycflights13::flights` tibble.
- ▶ Use `map()` to return the `summary()` of each column of the `nycflights13::flights` tibble.

## Pipes and `map()` functions

- ▶ Suppose we want to record a model summary returned by the `summary()` function.
  - ▶ `summary()` applied to an `lm()` object it computes regression summaries like standard errors and model  $R^2$ .

```
rr %>%  
  map(summary) %>%  
  map_dbl(function(ss) { ss$r.squared })
```

```
##           fit1           fit2           fit3           fit4  
## 0.78845184 0.91430933 0.73684218 0.04087594
```

- ▶ Notice that we can define a function on-the-fly in the call to a `map()` function.
- ▶ `map()` functions have a short-cut for function definitions.

```
rr %>%  
  map(summary) %>%  
  map_dbl(~.$r.squared) # or map_dbl("r.squared")
```

```
##           fit1           fit2           fit3           fit4  
## 0.78845184 0.91430933 0.73684218 0.04087594
```

- ▶ In `~.` read `~` as “define a function” and `.` as “argument to the function”

## Exercise

- ▶ Write a call to `map_dbl()` that does the same thing as `map_dbl(rr, b1fun)`, but define the function on the fly, as in the previous slide. You can use multiple calls to `map()` functions.

## Detour: The apply family of functions in R

- The “original” apply is `apply()`, which can be used to apply a function to rows or columns of a matrix.

```
mat <- matrix(1:6,ncol=2,nrow=3)
mat
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

```
apply(mat,1,sum) # row-wise sums; rowSums() is faster
```

```
## [1] 5 7 9
```

```
apply(mat,2,sum) # column-wise; colSums() is faster
```

```
## [1] 6 15
```

## Detour, cont.

- ▶ `sapply()` takes the output of `lapply()` and simplifies to a vector or matrix.

```
sapply(rr,coef)
```

##	fit1	fit2	fit3	fit4
## (Intercept)	0.0008831357	0.01572372	-0.06641184	-0.008394232
## x1	0.9281453769	1.03114836	0.04316514	-0.018428268
## x2	0.0426465892	1.00306653	0.93035180	-0.116309416

## Detour, cont.

- ▶ Other apply-like functions `vapply()`, `mapply()`, `tapply()`, ...
- ▶ I don't use these.
  - ▶ See their respective help pages for information.