Iteration

Tad Dallas

Contents

What is iteration?

Iteration refers to the process of doing the same task to a bunch of different objects. Consider a toy example of the actions required by a cashier at a grocery store. They scan each item, where items can be different sizes/shapes/prices. This is an iterative task, as it uses the same motions (essentially) across a variety of different objects (groceries) which may vary in many ways, but have some commonalities (e.g., most items have a barcode).

Why is iteration important?

Up until this point, we have dealt with single data.frame objects (or vectors, the building blocks of data.frames). However, we also introduced the concept of `lists` in one of the first lectures, and will go into more detail about lists soon. For now, we'll talk about iteration independent of list objects, but keep in mind that iteration is important for lists.

Essentially, iteration allows us to process a large amount of data without the need to repeat ourselves. Recall the gapminder data.

```
dat <- read.delim(file = "http://www.stat.ubc.ca/~jenny/notOcto/STAT545A/examples/gapminder/data/gapmin
```

We discussed the `gapminder` data when introducing some tools around data subsetting and summarising. We ended that lecture by discussing `dplyr`, a useful package for data processing.

```
library(plyr)
library(dplyr)
```

Recall that towards the end of that lecture, we introduce piping commands with `dplyr` to summarise data. For instance, the code below calculates mean life expectancy (`lifeExp`) by `country`.

```
tmp3 <- dat |>
    dplyr::group_by(country) |>
    dplyr::summarise(mnLifeExp=mean(lifeExp))
```

Approaching this with `dplyr` offers us a powerful way to summarise our data, but you will inevitably hit the limits of `dplyr` and thinking about how to do this in base R is difficult, right? In base R, we discussed subsetting, but to do what the above code does, we would have to subset by every country and then calculate the mean `lifeExp` for each subset. This is a good jumping off point for iteration, starting with the idea of the `for` loop (some folks use 'looping' and 'iteration' to mean the same thing). So we want a way to subset the `dat` data.frame by country, and then calculate mean `lifeExp`.

To start, we need to get a vector of the countries in the data.

```
countries <- unique(dat$country)
```

Then we need to get the overall structure of the loop in place. To do this, we use the structure `for(i in range){ do something}`. Essentially, we need to first define the range of what we want the loop to do, and then within the curly brackets, we need to do the thing. The power of this comes from the `i` in the `for` loop call. This is essentially saying to temporally treat `i` as one of the values in `range`, do something considering that, and then set `i` to the next value. This sequential process means that at the end of the loop, we will have cycled through all the entries in `range`.

```
for(i in countries){
  print(i)
}
```

```
## [1] "Afghanistan"
## [1] "Albania"
## [1] "Algeria"
## [1] "Angola"
## [1] "Argentina"
## [1] "Australia"
## [1] "Austria"
## [1] "Bahrain"
## [1] "Bangladesh"
## [1] "Belgium"
## [1] "Benin"
## [1] "Bolivia"
## [1] "Bosnia and Herzegovina"
## [1] "Botswana"
## [1] "Brazil"
## [1] "Bulgaria"
## [1] "Burkina Faso"
## [1] "Burundi"
## [1] "Cambodia"
## [1] "Cameroon"
## [1] "Canada"
## [1] "Central African Republic"
## [1] "Chad"
## [1] "Chile"
## [1] "China"
## [1] "Colombia"
## [1] "Comoros"
## [1] "Congo, Dem. Rep."
## [1] "Congo, Rep."
## [1] "Costa Rica"
## [1] "Cote d'Ivoire"
## [1] "Croatia"
## [1] "Cuba"
## [1] "Czech Republic"
## [1] "Denmark"
## [1] "Djibouti"
## [1] "Dominican Republic"
## [1] "Ecuador"
## [1] "Egypt"
## [1] "El Salvador"
## [1] "Equatorial Guinea"
```

```
## [1] "Eritrea"
## [1] "Ethiopia"
## [1] "Finland"
## [1] "France"
## [1] "Gabon"
## [1] "Gambia"
## [1] "Germany"
## [1] "Ghana"
## [1] "Greece"
## [1] "Guatemala"
## [1] "Guinea"
## [1] "Guinea-Bissau"
## [1] "Haiti"
## [1] "Honduras"
## [1] "Hong Kong, China"
## [1] "Hungary"
## [1] "Iceland"
## [1] "India"
## [1] "Indonesia"
## [1] "Iran"
## [1] "Iraq"
## [1] "Ireland"
## [1] "Israel"
## [1] "Italy"
## [1] "Jamaica"
## [1] "Japan"
## [1] "Jordan"
## [1] "Kenya"
## [1] "Korea, Dem. Rep."
## [1] "Korea, Rep."
## [1] "Kuwait"
## [1] "Lebanon"
## [1] "Lesotho"
## [1] "Liberia"
## [1] "Libya"
## [1] "Madagascar"
## [1] "Malawi"
## [1] "Malaysia"
## [1] "Mali"
## [1] "Mauritania"
## [1] "Mauritius"
## [1] "Mexico"
## [1] "Mongolia"
## [1] "Montenegro"
## [1] "Morocco"
## [1] "Mozambique"
## [1] "Myanmar"
## [1] "Namibia"
## [1] "Nepal"
## [1] "Netherlands"
## [1] "New Zealand"
## [1] "Nicaragua"
## [1] "Niger"
## [1] "Nigeria"
```

```
## [1] "Norway"
## [1] "Oman"
## [1] "Pakistan"
## [1] "Panama"
## [1] "Paraguay"
## [1] "Peru"
## [1] "Philippines"
## [1] "Poland"
## [1] "Portugal"
## [1] "Puerto Rico"
## [1] "Reunion"
## [1] "Romania"
## [1] "Rwanda"
## [1] "Sao Tome and Principe"
## [1] "Saudi Arabia"
## [1] "Senegal"
## [1] "Serbia"
## [1] "Sierra Leone"
## [1] "Singapore"
## [1] "Slovak Republic"
## [1] "Slovenia"
## [1] "Somalia"
## [1] "South Africa"
## [1] "Spain"
## [1] "Sri Lanka"
## [1] "Sudan"
## [1] "Swaziland"
## [1] "Sweden"
## [1] "Switzerland"
## [1] "Syria"
## [1] "Taiwan"
## [1] "Tanzania"
## [1] "Thailand"
## [1] "Togo"
## [1] "Trinidad and Tobago"
## [1] "Tunisia"
## [1] "Turkey"
## [1] "Uganda"
## [1] "United Kingdom"
## [1] "United States"
## [1] "Uruguay"
## [1] "Venezuela"
## [1] "Vietnam"
## [1] "West Bank and Gaza"
## [1] "Yemen, Rep."
## [1] "Zambia"
## [1] "Zimbabwe"
```

So what did the above code do?

Alright. So we have a way to sequentially work through all of the `countries` and we know how to subset the data based on country. So we can now subset the data for each of the countries, using the `i` iterator as a stand-in for each of the country names. But this does not actually do anything with the data, such that `tmp` will just be the subset data for the last country in the `countries` vector.

```
for(i in countries){
  tmp <- dat[which(dat$country == i), ]
}
```

So let's now compute the mean `lifeExp` for each country.

```
meanLifeExp <- c()
for(i in countries){
  tmp <- dat[which(dat$country == i), ]
  meanLifeExp <- c(meanLifeExp, mean(tmp$lifeExp))
}
```

Here, we first create a vector to hold the output data (`meanLifeExp`) and then append the value for each mean onto the vector. That is, we essentially re-write the `meanLifeExp` vector at every step of the iteration. This is bad practice for a number of reasons (e.g., no memory efficient, writing over objects where the object itself is in the call is bad practice, etc.). So how can we get around doing this? `for` loops can be handed a vector of character values (as we have done above) or they can be handed a numeric range. This is often useful, as it eases indexing and can be a bit clearer in the code.

```
meanLifeExp <- c()
for(i in 1:length(countries)){
  tmp <- dat[which(dat$country == countries[i]), ]
  meanLifeExp[i] <- mean(tmp$lifeExp)
}
```

And the results of this code should be the same as the other `for` loop. We now have a vector of mean life expectancy values for each country in `countries`. But that was a fair bit of work to get the same thing we could have gotten with `dplyr`, right? Let's explore a situation where it would be a bit tougher to get the same thing out of `dplyr` (at least with our current knowledge, as the example I'll give below can be solved using `dplyr::do`).

Let's say that we want to explore the relationship between `year` and `lifeExp` for each country. That is, we want to know how life expectancy is changing over time across the different countries. To do this, we can use the `cor.test` function in R to calculate Pearson's correlation coefficients (assumes linear structure between the two variables) or Spearman's rank correlation (assumes monotonic, but not linear response). The output of `cor.test` is a object, such that `dplyr::summarise` would fail.

```
tmp3 <- dat |>
    dplyr::group_by(country) |>
    dplyr::summarise(cor.test(year, lifeExp))
```

So summarise expects the output to be a vector (note that there are ways around this, by pulling out the information we want from the cor.test)

```
tmp3 <- dat |>
    dplyr::group_by(country) |>
    dplyr::summarise(cor.test(year, lifeExp)$estimate)
```

But how we do pull out multiple values from the same test? And how do we handle and diagnose potential errors when we don't work through each test sequentially?

```
lifeExpTime <- matrix(0, ncol=4, nrow=length(countries))

for(i in 1:length(countries)){
  tmp <- dat[which(dat$country == countries[i]), ]
  crP <- cor.test(tmp$year, tmp$lifeExp)
  crS <- cor.test(tmp$year, tmp$lifeExp, method='spearman')
  lifeExpTime[i, ] <- c(crP$estimate, crP$p.value,
```

```
    crS$estimate, crS$p.value)
}
```

```
## Warning in cor.test.default(tmp$year, tmp$lifeExp, method = "spearman"): Cannot
## compute exact p-value with ties
```

```
colnames(lifeExpTime) <- c('pearsonEst', 'pearsonP',
  'spearmanEst', 'spearmanP')

lifeExpTime <- as.data.frame(lifeExpTime)
lifeExpTime$country <- countries
```

And we can explore these data, to determine which countries have increasing or decreasing life expectancy values as a function of time.

```
lifeExpTime[which.min(lifeExpTime$pearsonEst),]
```

```
##     pearsonEst  pearsonP spearmanEst spearmanP country
## 141 -0.2446149 0.4435318  -0.1888112 0.5578278  Zambia
```

This may seem like a lot of work when we could have done a bit less using `dplyr` syntax. The real power of `for` loops will be in working with lists, simulating data, and plotting. For instace, let's say we don't have data directly to work with, but want to generate data. We could generate a bunch of data, mash it all together in a data.frame, and then feed it into `dplyr`, the data generation step would require a `for` loop already, so why not keep things all contained in the `for` loop.

Let's say we want to create a Fibonacci sequence. This is a vector of numbers in which each number is the sum of the two preceding numbers in the vector. For the example, we will limit the length of the vector to be length 1000.

```
fib <- c(0,1)
for(i in 3:1000){
  fib[i] <- sum(fib[(i-2):(i-1)])
}
```

And now we have a Fibonacci sequence starting with `c(0,1)`.

> Why do I start the `for` loop above at 3, and how else could you approach this same problem (there are many ways)?

Apply statements

`apply` statements exist in many types, depending on the data.structure you wish to do the action on: e.g. `apply`, `sapply`, `lapply`, `vapply`, `tapply`. We will focus on `apply` and `lapply`, but realize that these other options may be better suited for your use case (especially `vapply`, which gives you a bit more control over output format). In the loop above, we wanted to find the mean of each entry in a list. We used a `for` loop to loop over elements, and stored the resulting means in a vector called `out`. Instead, we could use `lapply`…the `l` in it means it performs some action on a list object.

```
lapply(X=testList2, FUN=mean)
```

```
## $a
## [1] 0.5149336
##
## $b
## [1] 0.7264648
##
## $d
## [1] 0.5308854
```

The output of `lapply` will always be a list, which is nice in some instances and not nice in others. `sapply` is a wrapper for `lapply` which always returns a vector of values.

```
sapply(X=testList2, FUN=mean)
```

```
##         a         b         d
## 0.5149336 0.7264648 0.5308854
```

Now that we have an idea of what the `apply` family of functions do, we can look specifically at `apply`, which operates on matrices or data.frames. What if we wanted to calculate the mean of every column or row in a data.frame? We could loop over each column or row…

```
testDF <- data.frame(a=runif(100), b=rpois(100,2), d=rbinom(100,1,0.5))


# over columns
ret <- c()
for(i in 1:ncol(testDF)){
    ret[i] <- mean(testDF[,i])
}


# over rows
ret <- c()
for(i in 1:nrow(testDF)){
    ret[i] <- mean(unlist(testDF[i, ]))
}
```

Or we could use apply statements

```
apply(X=testDF, MARGIN=2, FUN=mean)
```

```
##         a         b         d
## 0.5042976 1.9200000 0.5400000
```

```
apply(X=testDF, MARGIN=1, FUN=mean)
```

```
##   [1] 0.425184775 0.478412521 0.834622988 1.262198389 0.160172538 0.384958167
##   [7] 0.469749068 1.128428193 1.245687625 1.006998324 0.999168415 1.134846659
##  [13] 1.406664025 0.676674688 1.073979180 1.331710100 1.844174138 1.107557104
##  [19] 0.993917427 0.984007377 1.172282540 1.225771109 0.290384519 1.116905179
##  [25] 1.493889362 0.660497596 0.004541286 1.522423475 1.315873482 0.345797044
##  [31] 0.717075285 1.320199840 0.795332259 1.961740433 1.100990044 0.934668066
##  [37] 0.787430088 0.492282019 1.196640212 0.044324628 0.339410479 1.962883973
##  [43] 1.408847796 0.407031177 0.838931939 1.011333859 1.277959013 0.914728818
##  [49] 1.131055833 0.995985762 0.407513025 1.285932457 0.733941609 0.818124487
##  [55] 0.213124912 1.279590637 1.712525694 0.987830886 1.647503139 1.126318355
##  [61] 2.005576025 1.950797316 1.427232727 1.033746277 1.618414137 0.629113586
##  [67] 0.014085487 0.648695241 0.929680814 1.471100321 1.608327144 0.343905762
##  [73] 1.104362875 1.097495476 0.846468239 1.246504235 0.868313426 0.896822065
##  [79] 1.599102416 1.520846078 1.077551902 1.522040195 1.335592680 2.210423151
##  [85] 0.684558636 0.186544175 0.424142976 0.531286952 1.291440395 0.137399089
##  [91] 0.373592452 0.438049584 0.982093590 1.431807700 1.074423219 0.618701111
##  [97] 1.140794745 0.405091489 0.906829900 1.228233129
```

One advantage is that indexing rows of a data.frame is a pain, which is why we had to `unlist` each row in the for loop over rows above. If we do not do this, we get a vector of NA values. This is because a data.frame is a list of vectors. This is why column-wise operations on data.frames can also be performed using `lapply` (if we wanted list output) or `sapply` (if we wanted vector output).

```
lapply(X=testDF, FUN=mean)
```

```
## $a
## [1] 0.5042976
##
## $b
## [1] 1.92
##
## $d
## [1] 0.54
```

```
sapply(X=testDF, FUN=mean)
```

```
##         a         b         d
## 0.5042976 1.9200000 0.5400000
```

A fun class exercise

You are creating a game of rock-paper-scissors. In the game, each player can select their strategy, and the strategy can be different in each trial (where there can be 100s of trials).

I think that the outcome is random, so as a player, I already have decided what I'm going to play before the game starts.

```
strat <- sample(c('rock','paper', 'scissors'), 100, replace=TRUE)
```

Write a for loop to simulate rock-paper-scissors game of 500 trials between two players, where my strategy above is one of the players.

How would you go about changing the strategy of the other player to beat my strategy?

How would you modify your strategy to be adaptive? For instance, if your opponent selects 'rock' twice in a row, it may be unlikely that they'll select 'rock' again. How do you incorporate this into the code?

sessionInfo

```
sessionInfo()
```

```
## R version 4.3.1 (2023-06-16)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Ubuntu 22.04.3 LTS
##
## Matrix products: default
## BLAS:   /usr/lib/x86_64-linux-gnu/atlas/libblas.so.3.10.3
## LAPACK: /usr/lib/x86_64-linux-gnu/atlas/liblapack.so.3.10.3;  LAPACK version 3.10.0
##
## locale:
##  [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=en_US.UTF-8        LC_COLLATE=en_US.UTF-8
##  [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
##  [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
##  [9] LC_ADDRESS=C               LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C
##
## time zone: America/New_York
## tzcode source: system (glibc)
```

```
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
## [1] dplyr_1.1.2 plyr_1.8.8
##
## loaded via a namespace (and not attached):
##  [1] vctrs_0.6.3      cli_3.6.1        knitr_1.43       rlang_1.1.1
##  [5] xfun_0.39        generics_0.1.3   jsonlite_1.8.7   glue_1.6.2
##  [9] htmltools_0.5.5  tinytex_0.45     sass_0.4.7       fansi_1.0.4
## [13] rmarkdown_2.23   evaluate_0.21    jquerylib_0.1.4  tibble_3.2.1
## [17] fastmap_1.1.1    yaml_2.3.7       lifecycle_1.0.3  compiler_4.3.1
## [21] pkgconfig_2.0.3  Rcpp_1.0.11      digest_0.6.33    R6_2.5.1
## [25] tidyselect_1.2.0 utf8_1.2.3       pillar_1.9.0     magrittr_2.0.3
## [29] bslib_0.5.0      tools_4.3.1      cachem_1.0.8
```