Basics of Programming: if…else, iterations, custom functions

Enrico Toffalini

Conditional Programming

Conditional statements like `if`, `if...else`, and `ifelse` in R are essential tools for automating tasks and assisting decision making in data science. What follows are a few simple “toy examples”, but focus on the underlying logic. This will be greatly useful in more advanced applications

---

`if` statement

the `if` statement performs an action *only if* a condition is met

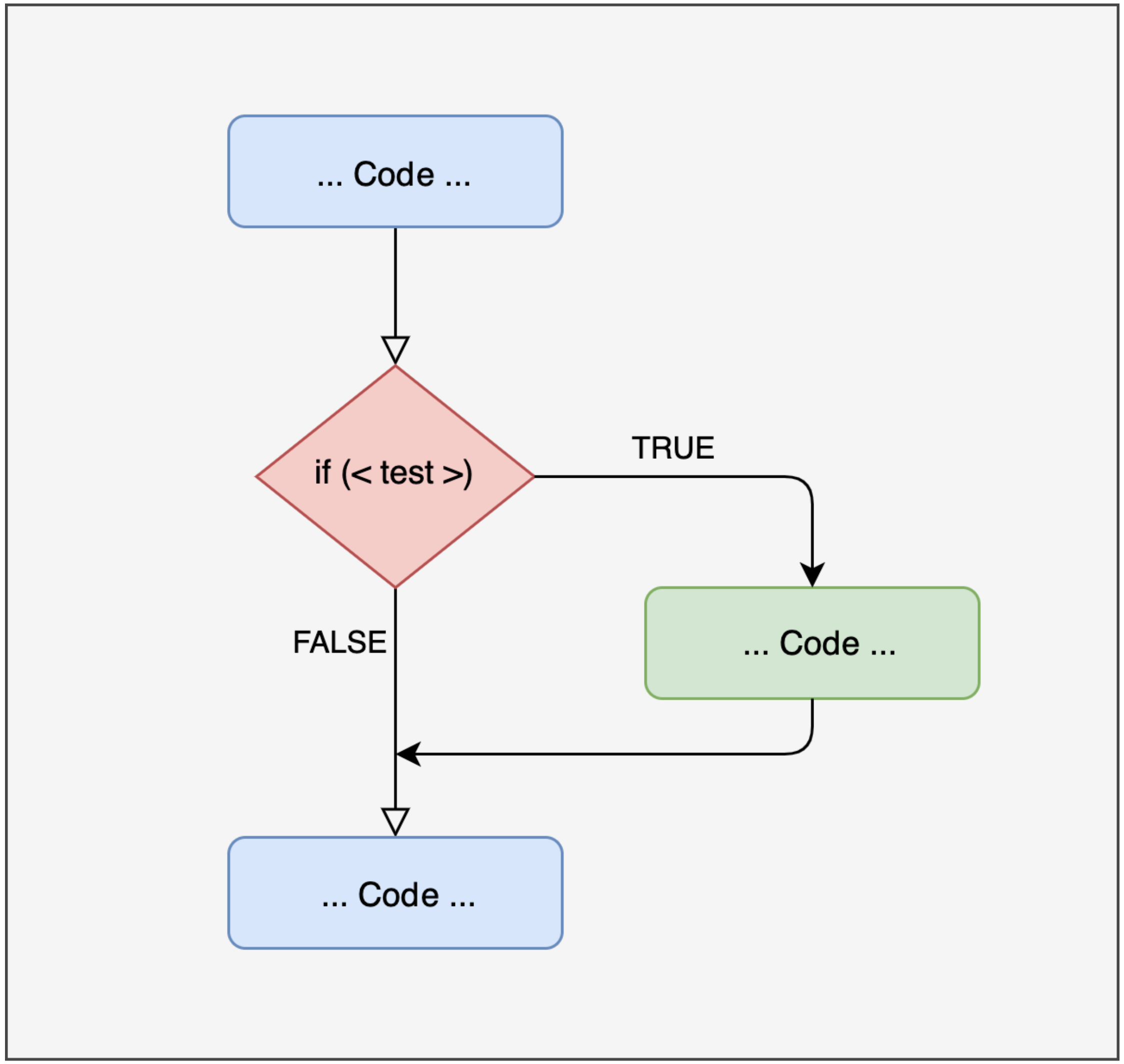
```r
age = 20
```

```r
if(age >= 18){
  print("Adult")
}
```

```
[1] "Adult"
```

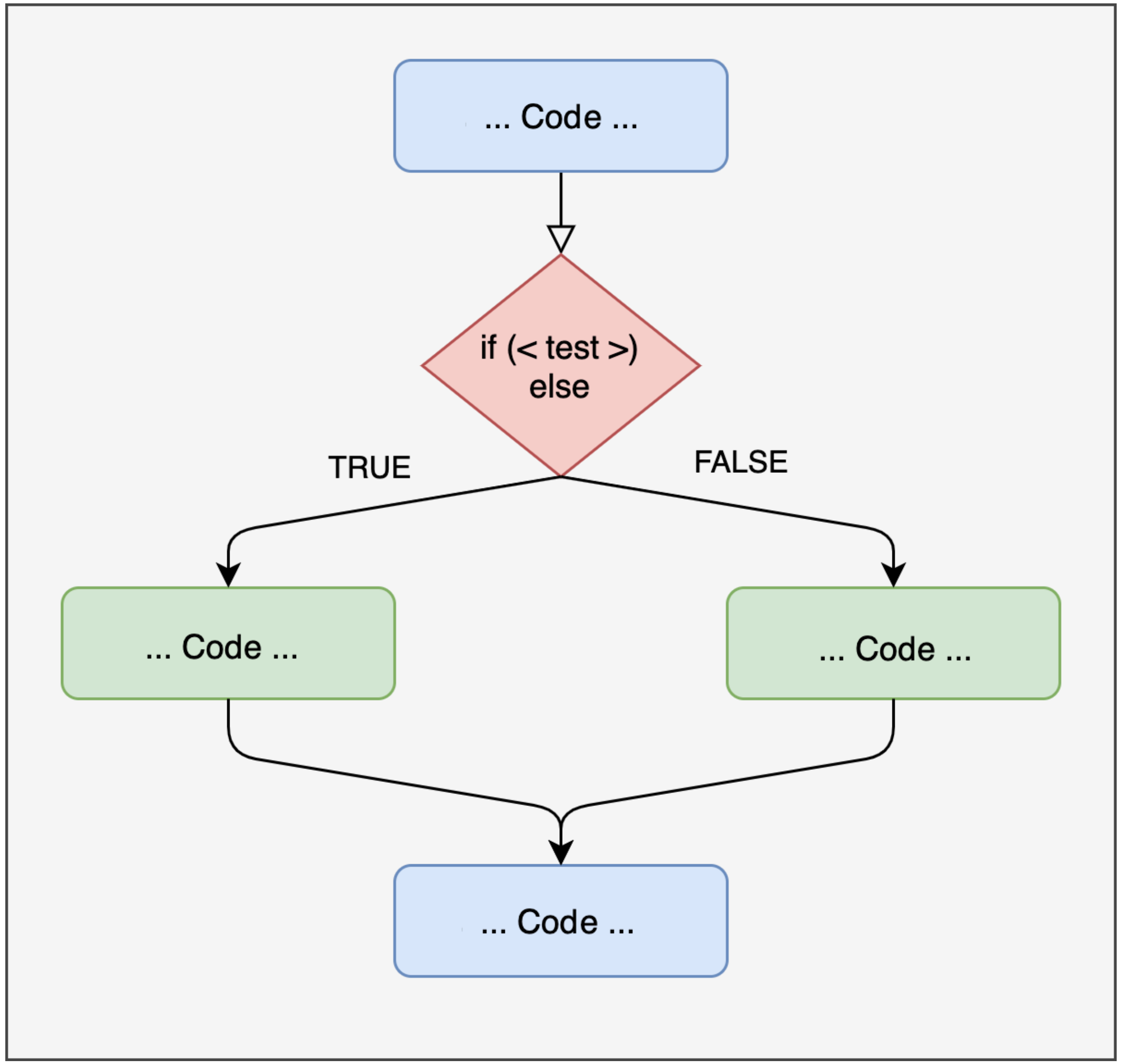
`if` statement

Basic flowchart showing the logic of the `if` statement:



`if...else` statement

Sometimes, however, you may need to perform *alternative* actions



`if...else` statement

Sometimes, however, you may need to perform *alternative* actions

Here is a practical example of the `if...else` statement

```
age = 15
```

```
if(age >= 18){
  print("Adult")
} else {
  print("Minor")
}
```

```
[1] "Minor"
```

In the above example:

- *if* `age` is 18 or older, R will print `"Adult"`;
- otherwise (*else*) it will print `"Minor"`

`if...else if...else` statement

When you need to evaluate more than just two alternative conditions, you can use **nested conditional statements**, that is you combine multiple `if...else` statements

```
age = 10
```

```
if (age >= 18) {
  print("Adult")
} else if (age >= 13) {
  print("Adolescent")
} else if (age >= 2){
  print("Child")
} else {
  print("Infant")
}
```

```
[1] "Child"
```

`if...else` statement

Possible, pratical use of `if...else` in a preplanned analysis for a hypothetical preregistered study: automate the decision to conduct additional analyses based on the result of a preliminary test. This helps create a reproducible analysis pipeline with a clear set of decisions

```r
## PREPLANNED ANALYSIS

# preliminary test
tt1 = t.test(x1, x2, data=df, paired=TRUE)

# based on the p-value of the preliminary t-test, choose the next step
if (tt1$p.value < 0.05) {
  # If significant, perform an additional analysis with a linear model (lm)
  print("Significant result: proceeding with follow-up analysis")
  fit = lm(outcome ~ pred1 + pred2 * moder1, data = df)
  summary(fit)
} else {
  # else, report only the preliminary test
  print("No significant result: reporting preliminary results only")
  print(tt1)
}
```

`if...else` statement

All previous examples required to evaluate a single particular condition that might be `TRUE` or `FALSE`. However, you often want to apply this type of operation to an entire vector

Using `if` and `if...else` directly on a vector will **NOT** work as intended:

```
age = c(2, 28, 15, 1, 4, 67, 42, 14, 7)

if(age >= 18){
  print("Adult")
} else {
  print("Minor")
}
```

```
Error in if (age >= 18) {: the condition has length > 1
```

`ifelse()`

All previous examples required to evaluate a single particular condition that might be `TRUE` or `FALSE`. However, you often want to apply this type of operation to an entire vector

To handle such cases you can use the base **`ifelse()`** function, that evaluates each element of a vector individually:

```
age = c(2, 28, 15, 1, 4, 67, 42, 14, 7)

ifelse(age >= 18, "Adult", "Minor")
```

```
[1] "Minor" "Adult" "Minor" "Minor" "Minor" "Adult" "Adult" "Minor"
"Minor"
```

— *Best practices: store results and set type as appropriate!*

```
ageCategory = ifelse(age >= 18, "Adult", "Minor")
ageCategory = factor(ageCategory, levels=c("Minor", "Adult"))
ageCategory
```

```
[1] Minor Adult Minor Minor Minor Adult Adult Minor Minor
Levels: Minor Adult
```

ifelse()

The `ifelse()` function can also be nested to manage multiple conditions, such as in the following example:

```
age = c(2, 28, 15, 1, 4, 67, 42, 14, 7)

ifelse(age >= 18, "Adult" ,
                ifelse(age >= 13, "Adolescent" ,
                              ifelse(age >= 2, "Child",
                                                  "Infant")))
```

```
[1] "Child"      "Adult"      "Adolescent" "Infant"     "Child"
[6] "Adult"      "Adult"      "Adolescent" "Child"
```

`dplyr::case_when()`

While this works, the nested structure can become cumbersome as the number of conditions increases.

The above case may become cumbersome and less readable when you need to combine a large number of conditions. In such cases, the **`case_when()`** function from the `dplyr` package (part of the *tidyverse* collection)

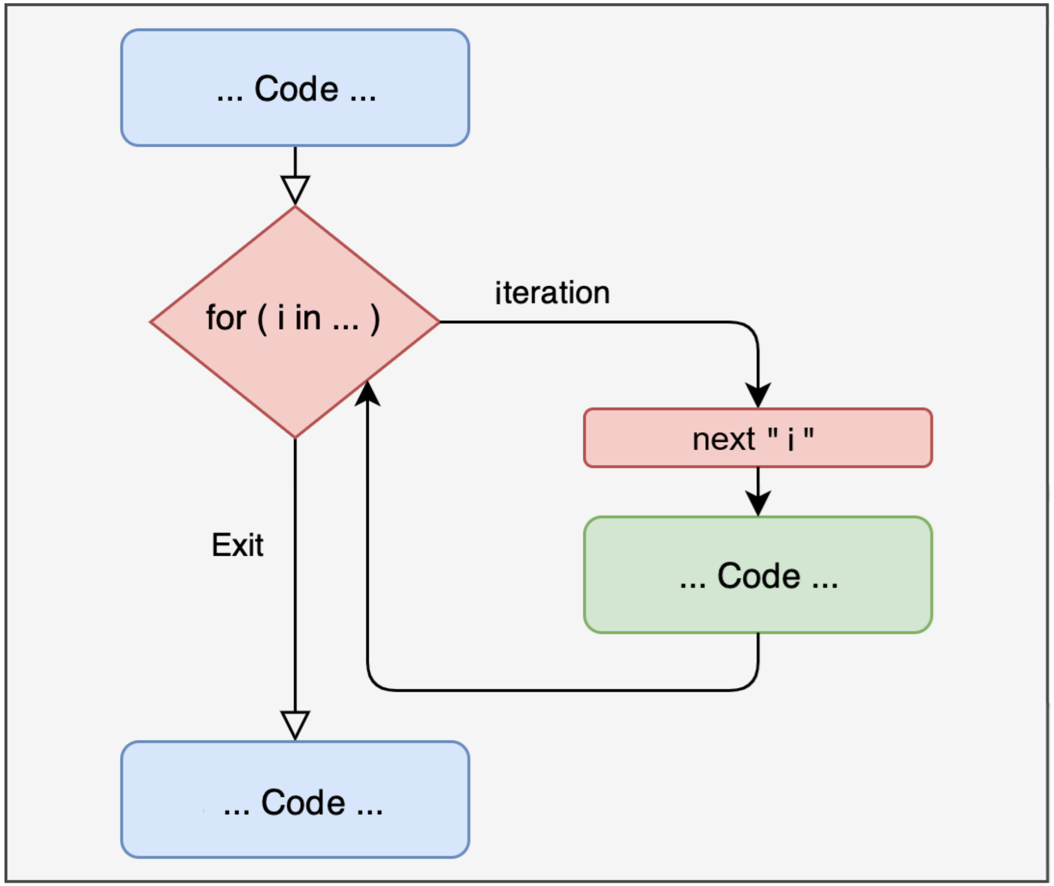
```r
age = c(2, 28, 15, 1, 4, 67, 42, 14, 7)

library(dplyr)

case_when(
  age >= 18 ~ "Adult",
  age >= 13 ~ "Adolescent",
  age >= 2  ~ "Child",
  TRUE      ~ "Infant"
)
```

```
[1] "Child"      "Adult"      "Adolescent" "Infant"     "Child"
[6] "Adult"      "Adult"      "Adolescent" "Child"
```

Iterative Programming

Iterative programming allows you to repeat one or a series of actions automatically, for a predetermined number of times or until a condition is met

Here's the basics of iterative programming with the `for` loop:



for loop

Here are a few simple examples of using the `for` loop

```
for(i in 1:5)  print(i)
```

```
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

```
for(i in 1:5)  print(i^2)
```

```
[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
```

```
for(i in 1:5){
  print(Sys.time())
  Sys.sleep(1)
}
```

```
[1] "2024-11-23 14:46:26 CET"
[1] "2024-11-23 14:46:27 CET"
[1] "2024-11-23 14:46:28 CET"
[1] "2024-11-23 14:46:29 CET"
[1] "2024-11-23 14:46:30 CET"
```

```
for(i in 1:5){
  print(Sys.time())
  Sys.sleep(2)
}
```

```
[1] "2024-11-23 14:46:31 CET"
[1] "2024-11-23 14:46:33 CET"
[1] "2024-11-23 14:46:35 CET"
[1] "2024-11-23 14:46:37 CET"
[1] "2024-11-23 14:46:39 CET"
```

`for` loop

Here's a more interesting example of iterative `for` loop with practical usefulness: we want to repeat a data simulation for a predetermined number of times (5 iterations), each time drawing $n = 30$ values from a *standard normal distribution*, computing and displaying the average …

```
set.seed(0) # set a seed for reproducibility: best practice!

for(i in 1:5){
  x = rnorm(n = 30, mean = 0, sd = 1)
  print(mean(x))
}
```

```
[1] 0.02195079
[1] -0.02577153
[1] -0.009581231
[1] 0.03212316
[1] -0.2946441
```

This is actually the starting point of a Monte Carlo simulation! 😀

for loop

in the previous example, the `for` loop displayed the results but didn't store it. For more effective use, you can combine the `for` loop with indexing with `[]` to save each result:

```
set.seed(0) # set a seed for reproducibility: best practice!
niter = 5 # set the desired number of iterations: best practice!
# initialize a results vector with NAs: best practice!
results = rep(NA, niter)
# now run the for loop! :-)
for(i in 1:niter){
  x = rnorm(n = 30, mean = 0, sd = 1)
  results[i] = mean(x)
}
results # display results
```

```
[1]  0.021950789 -0.025771530 -0.009581231  0.032123159 -0.294644080
```

```
sd(results) # estimate standard error of the mean
```

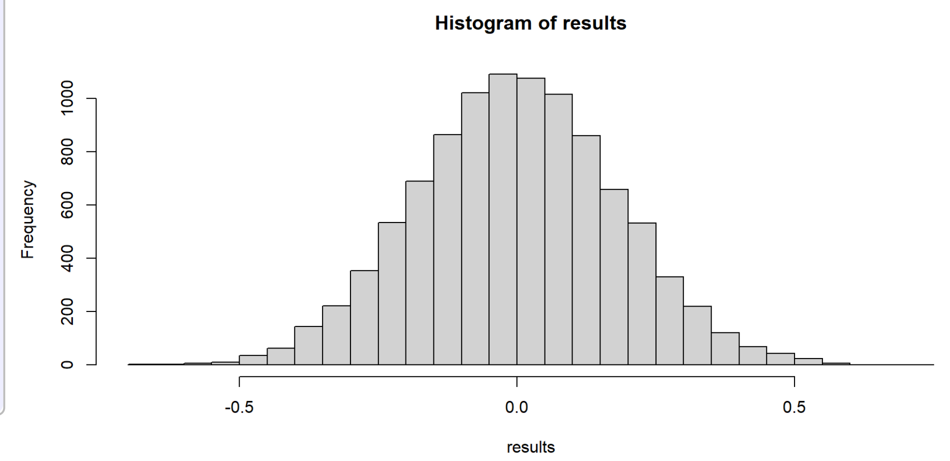
```
[1] 0.1358843
```

for loop

Let's extend the previous example with … a few more iterations!

```r
# STEP 1: RUN SIMULATION

# set number of iterations
niter = 10000
# initialize results vector
results = rep(NA, niter)
# actually run simulation
for(i in 1:niter){
  x = rnorm(n = 30, mean = 0, sd = 1)
  results[i] = mean(x)
}
```

```r
# STEP 2: PLOT RESULTS

hist(results, breaks=50)
```




```r
# STEP 3: COMPUTE SD OF RESULTS
sd(results)
```

```
[1] 0.1806616
```

→ Enjoy it! This is a proper estimation of the *Standard Error of the Mean* via Monte Carlo simulation! 😊

`for` loop

You don't necessarily have to iterate over a sequence of integers (e.g., "`i in 1:10000`"; "`j in 1:ncol(df)`"), although this is the most common practice. You could iterate over whatever, for example, directly over the elements of a vector or other data structures

```
myVect = c("this", "is", "a", "vector", "of", "strings")

for(x in myVect){
  print(toupper(x))
}
```

```
[1] "THIS"
[1] "IS"
[1] "A"
[1] "VECTOR"
[1] "OF"
[1] "STRINGS"
```

`while` loop

The `while` loop is another type of iterative structure in R. It may be useful when the precise number of iterations is *not* predetermined, but depends on a target being reached

```r
amount = 1000
month = 0
interest_rate = 0.001 # 0.1% monthly interest rate

while(amount < 1500){
  month = month + 1
  amount = amount + amount * interest_rate
}

month
```

```
[1] 406
```

Interpretation: it takes 406 months to reach an amount of €$1,500$ *when starting with an amount of* €$1,000$ *with a* 0.1% *monthly interest rate*

`repeat` loop

The `repeat` loop has a logic similar to the `while` loop but 1) it always runs at least one iteration, 2) It explicitly emphasizes repetition until a condition (not necessarily a target) is met, using a `break` statement to terminate

```r
repeat{
  roll_die = sample(1:6, 1)
  print(roll_die)
  if(roll_die == 6) break
}
```

```
[1] 2
[1] 3
[1] 1
[1] 3
[1] 1
[1] 1
[1] 5
[1] 6
```

```r
attemptedExper = 0

repeat{
  attemptedExper = attemptedExper + 1
  tt = t.test(rnorm(10),rnorm(10))
  if(tt$p.value < 0.05) break
}

# number of experiments I attempted to...
# get a false positive :-)
attemptedExper
```

```
[1] 12
```

```r
round( tt$p.value , 3 )
```

```
[1] 0.041
```

`apply` family

`apply` is a family of base functions that provide **efficient tools** for running iterations on structures like dataframes, vectors, matrices, lists

Traditional loops provide a straghtforward, intuitive way to compute sequences of operations, but the `apply` **family allows you to run faster computations**… this may become particularly important when you need to *parallelize* for computationally intensive tasks

The following is *not* a computationally heavy task — but for example, let's say we want to compute the mean value *per column* in this dataframe:

```
   BD SI DS PCn CD VC LN MR CO SS
1  13 10  7  10 15  7 10 16  8 13
2   7 11  6   8 13 10  9  5  9 14
3  12  6  5   7  9  7  7  6  9  8
4   8  7  9  11  1  5  7  6  8  4
5  12 13  8  10 10 10  9 11 13 12
6  13 17 13   7 10 19 13 10 15 13
7  12 10  9   5 10  8  7  9 11 11
8   9 12 15  14  7 11 14 13  8 14
9  11 14  8  11  8 12 14 12 10  9
```

```
10 13 12 14  11  5 15 17 14 14  8
11  7  7  7   6  6  6  4  7 12  9
12 10 11  8   8 10  7  7  8  8 15
```

apply family

Here is how you could use the base `apply` **function** for computing the *mean* value by column:

```
apply(df, MARGIN=2, FUN=mean, na.rm=T)
```

```
       BD        SI        DS       PCn        CD        VC        LN        MR
 9.824121  9.856423  9.706767  9.889169  9.781955  9.867168  9.987437  9.904762
       CO        SS
 9.889447 10.005051
```

In fact, for such a simple task, even `colMeans()` could be sufficient:

```
colMeans(df, na.rm=T)
```

```
       BD        SI        DS       PCn        CD        VC        LN        MR
 9.824121  9.856423  9.706767  9.889169  9.781955  9.867168  9.987437  9.904762
       CO        SS
 9.889447 10.005051
```

but let consider slightly more complex cases …

apply family

Let's say you need to compute the *standard deviation* per column …

```
apply(df, MARGIN=2, FUN=sd, na.rm=T)
```

```
      BD       SI       DS      PCn       CD       VC       LN       MR 
2.941790 3.137753 3.072283 2.855583 3.022541 3.167819 2.951726 2.989253 
      CO       SS 
2.999217 2.896523 
```

… or to count the number of NA occurrences per column

```
apply(df, MARGIN=2, FUN=function(x) sum(is.na(x)) )
```

```
 BD  SI  DS PCn  CD  VC  LN  MR  CO  SS 
  2   3   1   3   1   1   2   1   2   4 
```

in the latter case, we had to define a custom function, but that's relatively simple to do!

`apply` family

Although any of such tasks could be done using a `for` loop, the code would be more cumbersome and less efficient. For example, here's how the exact same result as the latter `apply` example could be obtained using a `for` loop:

```
results = rep(NA, ncol(df))
names(results) = colnames(df)

for(i in 1:ncol(df)){
  results[i] = sum(is.na(df[,i]))
}

results
```

```
 BD  SI  DS PCn  CD  VC  LN  MR  CO  SS
  2   3   1   3   1   1   2   1   2   4
```

`apply` family

FYI, other functions within the `apply` **family**:

- **`tapply()`**: applies a function to subsets of a vector grouped by a factor, example `tapply(df$values, df$group, FUN=mean)` (know that the function `aggregate()` might be more convenient in some cases)
- **`lapply()`**: applies a function to each element of a *list*, returning results in a *list* format, example `lapply(my_list, length)`
- **`sapply()`**: the same as the previous one but returns results as a *vector* if possible, example `sapply(my_list, length)`
- **`mapply()`** multivariate version of `sapply()` that runs across more lists or vectors

```r
mapply(rnorm, n=c(2, 6, 4), mean=c(0, 100, 200), sd=c(1, 15, 30))
```

```
[[1]]
[1] -0.08178004  2.88841825

[[2]]
[1] 117.59238 108.24465 111.68657  91.52303 101.54431 104.22282

[[3]]
[1] 185.7671 186.2599 215.0463 187.8461
```

`sapply`

Here's how **`sapply()`** can be used to help compute the *standard error of the mean* via *Monte Carlo simulation*

```
myList = list()
for(i in 1:10000) myList[[i]] = rnorm(30)

ms = sapply(myList, mean)

sd(ms)
```

```
[1] 0.1851159
```

1. an empty list is initialized;

2. a `for` loop is used to fill each slot in the list with a vector of 30 randomly generated numbers;

3. `sapply` is used to compute the `mean` of each vector in the list;

4. standard error of the mean is computed as the `sd` on the vector of the means

Saving all generated data allows for more flexibility in analysis, although it uses more memory

`lapply` and `sapply`

Here's another, even more compact way of doing the same, without using any `for` loop, but this requires defining a small custom function:

```
results = lapply(1:10000, function(i) rnorm(30) )

ms = sapply(results, mean)

sd(ms)
```

```
[1] 0.1836006
```

Note: `"i"` represents the current element of `1:10000` over which `lapply` iterates. Even though it is practically useless here, because only random numbers are generated at each iteration, it must be included because `sapply` must pass an alement as the argument to the function by default

Define Custom Functions with `function()`

Previously, we saw a few cases of custom functions, for example for counting `NA`s in vectors, or computing the mean of a randomly generated vector

Here is the schema for defining custom functions with input (argument[s]), body, and output (return):

```
# define new function with some argument(s) as input
myFunctName = function(arg1 = NA, arg2 = NA, arg3 = TRUE){

  # body, series of operations
    computes several operations
    variously uses arg1, arg2, arg3
    get an object named "res" as final result

  # gives an output
  return(res)
}
```

Define Custom Functions with `function()`

Here's a full example:

```
z_score = function(vect = NA){

  vectM = mean(vect, na.rm=T)
  vectSD = sd(vect, na.rm=T)
  z = (vect - vectM) / vectSD

  return(z)
}
```

After creating it, a custom function can be used like any other function:

```
x = c(101, 90, NA, NA, 114, 87, 106, 98, 93)
z_score(x)
```

```
[1]  0.27162785 -0.89033574          NA          NA  1.64485756 -1.20723491
[7]  0.79979313 -0.04527131 -0.57343658
```

Define Custom Functions with `function()`

Here's a slightly more sophisticated example:

```
z_score = function(vect = NA, naRemove = FALSE){

  vectM = mean(vect, na.rm=naRemove)
  vectSD = sd(vect, na.rm=naRemove)
  z = (vect - vectM) / vectSD

  return(z)
}
```

Let's use it:

```
x = c(101, 90, NA, NA, 114, 87, 106, 98, 93)
z_score(x)
```

```
[1] NA NA NA NA NA NA NA NA NA
```

```
z_score(x, naRemove=TRUE)
```

```
[1]  0.27162785 -0.89033574          NA          NA  1.64485756 -1.20723491
[7]  0.79979313 -0.04527131 -0.57343658
```

Define Custom Functions with `function()`

In some previous examples, custom functions were used directly in combination with `apply()`, without curly brackets `{}` or `return()`, yet they worked!

Why?

Generally, curly brackets `{}` and `return()` enhance clarity and are best practice, but in some cases they can be omitted for more compact code:

- Curly brackets `{}` can be skipped if all code fits on a single line
- `return()` can be omitted if the last (or only) code line represents the output

Define Custom Functions with `function()`

To clarify the previous slide, these are **four alternative and increasingly compact ways** of writing the same function:

```
naCount = function(vect){
  whereNAs = is.na(vect)
  totalNAs = sum(whereNAs)
  return(totalNAs)
}

naCount = function(vect){
  whereNAs = is.na(vect)
  totalNAs = sum(whereNAs)
  totalNAs
}

naCount = function(vect){
  sum(is.na(vect))
}

naCount = function(vect) sum(is.na(vect))
```