R Introduction - Basics of Programming: for, if, functions, vectorizations

Jana Jarecki, Stephan Lewandowsky

10 July 2022

Agenda: What today teaches or repeats

if and *for*

- How to use *if* statements
- When == is useful
- How to use *for* loops
- When *for* loops are useful
- Breaking and skipping loops

Functions

- How to write your own functions
- How the scope of a function works

Vectorization

- When not to use *for* loops
- How to apply a function with `sapply()`
- How to apply a function with `apply()`

Fun R Facts - Mini sections with fun unique things about R

Fun R Facts About `round()`

```r
round(1.5)
```

```r
round(2.5)
```

Interesting R Facts About `round()`

What happened here?

- For rounding off a 5, the IEC 60559 standard, defined as *go to the even digit*
- This is a more fair rounding standard
- See the help for `?round`

But wait … ?

If you need to round up a 5, then you need to write your `round2()` function

If and for

`if()` conditions

- *if* statements implement conditions such as “if the prediction equals the observation, do A, else do B”

- *if* statements have two steps (1) compute a logical condition that can be `TRUE` or `FALSE` (2) execute a consequence if the condition is `TRUE`

```
observation <- 0
z <- NA
if (observation == 1) {
  z <- 1
} else {
  z <- 0
}
z
```

```
## [1] 0
```

Logical Statements

Logical statements usually are comparisons for exact equalitites and/or inequalities

```
x == y
x != y
x < y
x <= y
x > y
x >= y
```

Here's one example

```
x <- 0.15
x == 0.15
```

```
## [1] TRUE
```

```
x != 0.15
```

```
## [1] FALSE
```

Exercise: An *if* statement

If one random draw from a normal distribution *M=1, SD=2* is at least equal to 2 then assign `1` to a variable, otherwise assign the value of the draw, except when the value is lower than -2, then assign 0 to the variable.

Exercise: An *if* statement

```r
b <- 2
x <- rnorm(1, m=1, sd=2)
if (x >= b) {
  var <- 1
} else {
  if (x <= -b) {
    var <- 0
  } else {
    var <- x
  }
}
var
```

```
## [1] 1.386425
```

Fun R Facts About ==

Interesting example A

```r
x <- 0.1
x <- x + 0.05
x == 0.15
x != 0.15
```

Interesting example B

```r
a <- sqrt(2)
a * a == 2
a * a != 2
```

What happened here?

What happened here?

Fun R Facts: What happened?

- Not all numbers in R can be represented exactly because of so-called floating point precision
- For more information, see the R FAQ 7.31
- See also the Appendix G in this article on the issue, for some examples

Possible Solution(s)

Check out `?all.equal()` or round the numbers to a certain precision.

```r
all.equal(x, 0.15)
```

```
## [1] "Mean relative difference: 0.891808"
```

```r
round(x, digits=10) == 0.15
```

```
## [1] FALSE
```

```r
eps <- 0.000000000001
(x - 0.15) < eps
```

```
## [1] FALSE
```

Chain logical statements together

```
(x == y) & (a > b)
(x == y) | (a > b)
(x == y) || (a > b) # don't do thos
(x == Y) && (a > b) # don't do this
```

Note the bracket notation: `() & ()`

Fun R Facts: & versus &&

What happens here?

```r
a <- -2:2
print(a)
```

```
## [1] -2 -1  0  1  2
```

```r
(a >= 0) & (a <= 0)

(a >= 0) && (a <= 0)
```

The same applies to |, ||

What happens here?

```r
b <- 0:4
print(b)
```

```
## [1] 0 1 2 3 4
```

```r
(b >= 0) & (b <= 0) # don't do this

(b >= 0) && (b <= 0) # don't do this
```

The same applies to |, ||

`for()` loops

`for()` Loops

- Most programs have to repeat executions
- Such as *for each model: do something* or *for each participant: do something*, etc.

```
values <- seq(0,1,.1)
output <- NULL
for (v in values) {
  output <- c(output, 2^v)
}
output
```

```
##  [1] 1.000000 1.071773 1.148698 1.231144 1.319508 1.414214 1.515717 1.624505
##  [9] 1.741101 1.866066 2.000000
```

The Principle of `for()`

- *for* statements implement repeated execution of code

```
# for loop
for (v in value_vector) {
  # body of the loop: do something with v
  # where v takes on each value in the value_vector
}
```

Exersises: Simple *for* loop

- Compute the square root for each element in a vector `v` such that the results are stored in `v`, using a *for* loop.
- Compute the mean across rows of a matrix `M`, using *for* loops. Compute it also across columns.

```
v <- c(2, 5, 10, 23)
M <- matrix(data=rep(1:7, times=7) + rnorm(49), nrow=7)
```

Exersises: Simple *for* loop

```
for (i in 1:4) {
  v[i] <- sqrt(v[i])
}
print(v)
```

```
## [1] 1.414214 2.236068 3.162278 4.795832
```

```
rowm <- colm  <- NULL
for (i in 1:nrow(M)) {
  rowm[i] <- mean(M[i, ])
}
for (i in 1:ncol(M)) {
  colm[i] <- mean(M[, i])
}
print(rowm)
```

```
## [1] 1.170851 1.670133 3.052657 3.789746 5.195861 6.560312 6.924305
```

Exercise: A More Meaningful *for* Loop

Compute the difference between `predictions` and `observation` and store the absolute value of these differences, using a *for* loop. Print the mean of the result.

(Optional) Compute the joint likelihood, which means the product of p(obs | pred), of the observations given the predictions in a *for* loop.

```r
predictions <- c(.76, .89, .12, .34, .50)
observations <- c(0, 1, 0, 0, 1)
```

```r
y <- NULL
for (i in 1:length(predictions)) {
  y[i] <- abs(predictions[i] - observations[i])
}
print(mean(y))
```

```
## [1] 0.366
```

Nesting *for* loops

Loops can be nested

```
M <- matrix(NA, nrow=4, ncol=4)
for (i in 1:4) {
  for (j in 1:4) {
    M[i, j] <- i * j
  }
}
print(M)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    2    4    6    8
## [3,]    3    6    9   12
## [4,]    4    8   12   16
```

Exercises: Let's Nest Some *for* Loops

Replace the values in the matrix `M` with its square, using nested *for* loops.

(Optional)Z-standardize the values in `M` across rows using nested for loops. Repeat across columns.

```
M <- matrix(data=rep(1:7, times=7) + rnorm(49), nrow=7)
print(M)
```

```
##          [,1]     [,2]     [,3]      [,4]       [,5]      [,6]     [,7]
## [1,] 2.261516 1.277234 2.830764 0.5700752 -0.3726514 -1.050183 1.639941
## [2,] 1.761564 2.970719 0.894695 1.8674243  1.5206000  1.449166 1.951482
## [3,] 1.002187 3.952631 2.029521 4.0314958  3.0110135  2.420725 2.565655
## [4,] 4.701076 3.603047 2.666699 4.0069714  4.6287441  4.528144 3.717687
## [5,] 6.220819 4.997805 5.747007 5.4334090  6.9760633  4.860246 6.339730
## [6,] 5.149076 5.633075 4.786621 7.2873693  6.4615582  7.545938 7.046641
## [7,] 6.436392 7.039895 7.726008 4.5817357  7.5260161  8.173737 7.749215
```

```
for (i in 1:nrow(M)) {
  for (j in 1:ncol(M)) {
    M[i, j] <- M[i, j]^2
  }
}
print(M)
```

for makes particularly sense with iterative computations

Iterative computations?

- Iterative: when the *i + 1*. result is a function of the *i*. result

Iterative problems could be, for example …

- Find a value by some percentage better than your last value (iterate over values)
- Optimize decision trees from the ultimate goal to the current choice (backward induction)
- Learn something in time `t + 1` that depends on an experience in time `t` (iterate over times)

for makes particularly sense with iterative computations

```
x <- 1:5
for (t in 1:5) {
  y[t] <- y[t] + x[t-1]
}
```

There are some more exersises and examples for iterative computations in the very last section at the very end of this presentation if you are interested.

Functions `function()`

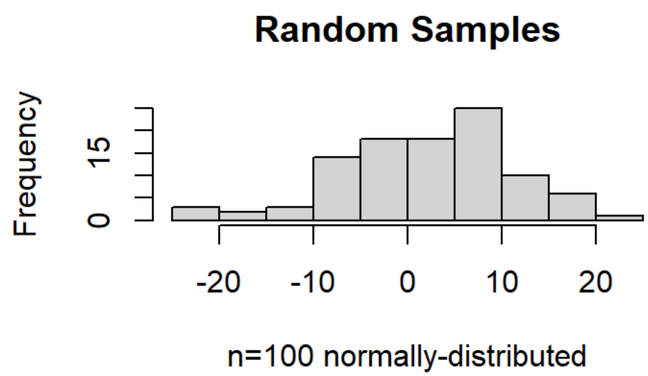
Functions

R is largely based on functions

```r
sqrt(2)
```

```
## [1] 1.414214
```

```r
hist(x = rnorm(n = 100, mean = 2, sd = 10),
     xlab = "n=100 normally-distributed",
     main = "Random Samples")
```



Our own functions

- Any meaningful program/model will (and should!) contain functions
- A cognitive model is a (sometimes complex) function that maps inputs to outputs
- We can write our own functions according to the following rules

```r
nameoffunction <- function(argument1, argument2) {
  # body of function, does interesting stuff
  # Returns a result called y
  # return(x)
}
```

Exercise: Our own functions

Rewrite the “logistic” assignment using one or more function(s).

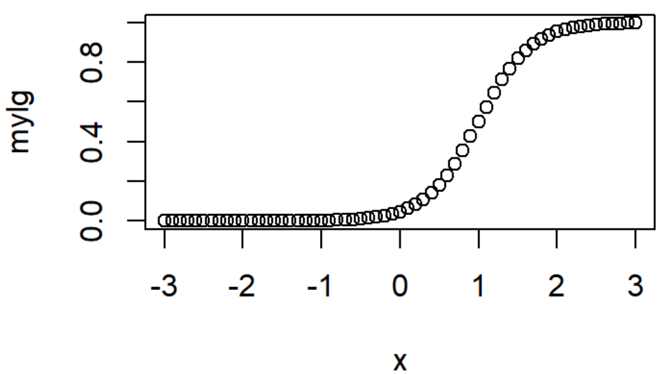
Define an object `done <- 0` before your function. From within the function, assign `1` to `done`. Tipp: you may need to google here.

```r
x <- seq(from=-3, to=3, by=0.1)
b <- 1
a <- 0
y <- 1/(1 + exp(-(x - a) * b))

nameoffunction <- function(argument1, argument2) {
  # body of function does interesting stuff
  # return(y)
}
```

Our own logistic function

```r
mylogistic <- function(x, a, b) {
  y <- 1/(1 + exp(-(x - a) * b))
  return(y)
}
x <- seq(from=-3, to=3, by=0.1)
mylg <- mylogistic(x=x, a=1, b=3)
plot(x=x, y=mylg)
```




```r
done <- 0
mylogistic <- function(x, a, b) {
  y <- 1/(1 + exp(-(x - a) * b))
  done <<- 1
  # or: assign("done", 1, envir = .GlobalEnv)
  return(y)
}
mylogistic(x=1, a=1, b=3)
```

```
## [1] 0.5
```

```r
print(done)
```

```
## [1] 1
```

Vectorization

Vectorization: Efficiency and elegance

- *for* loops are an essential concept
- Most programs/cognitive models contain many loops
- In R, loops can often (but not always) be replaced by vectorization

Vectorization: From *for* to elegance

for loops are not vectorized

```r
v <- c(2, 5, 10, 23)
for (i in 1:4) {
  v[i] <- sqrt(v[i])
}
print(v)
```

```
## [1] 1.414214 2.236068 3.162278 4.795832
```

$$\begin{pmatrix} 2 \\ 5 \\ 10 \\ 23 \end{pmatrix}$$

execute a function *f(x)* one by one for one element of a vector

$$\begin{pmatrix} f(2) \\ f(5) \\ f(10) \\ f(23) \end{pmatrix}$$

Code that is vectorized

```r
v <- c(2, 5, 10, 23)

v <- sqrt(v)

print(v)
```

```
## [1] 1.414214 2.236068 3.162278 4.795832
```

$$\begin{pmatrix} 2 \\ 5 \\ 10 \\ 23 \end{pmatrix}$$

execute a function *f(x)* at once for all elements of a vector

$$f\begin{pmatrix} 2 \\ 5 \\ 10 \\ 23 \end{pmatrix}$$

We have already used vectorization

- Vectorization is so intuitive, we have used it without worrying about it

```r
x <- c(7, 6, 1, 2, 0, -1, 4, 3, -2, 0)  #from before
x <- x + 3
```

- But we can take it (a lot) further using R functions that replace loops

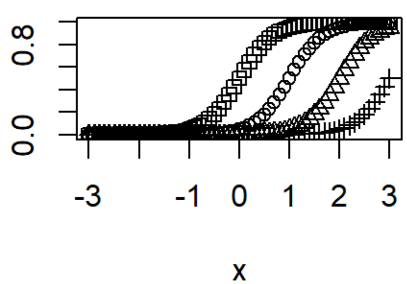
```r
# Execute FUN for each X
sapply(X, FUN)                   # if X is a vector
lapply(X, FUN)                   # if X is a list
apply(X, MARGIN, FUN)            # if X is a matrix
# MARGIN=1 if X are rows, MARGIN=2 if Xs are columns

tapply(X, INDEX, FUN)            # X is e.g. a vector or list
# INDEX is a factor with groups such as treatment or gender
```

The *apply* family

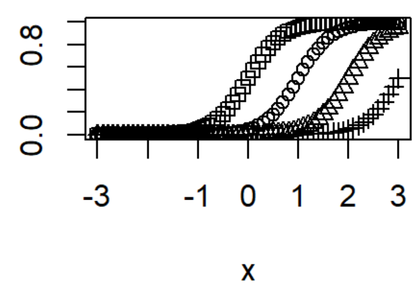
for loop - not vectorized

```
x <- seq(from=-3, to=3, by=0.1)
mylg <- mylogistic(x=x, a=1, b=3)
plot(x=x, y=mylg, ylab="", main="")
a_values <- c(0, 2, 3)
# for
for (a in a_values) {
  points(x=x,
         y=mylogistic(x=x, a=a, b=3),
         pch=a) #pch: point character
}
```



apply - vectorized

```
x <- seq(from=-3, to=3, by=0.1)
mylg <- mylogistic(x=x, a=1, b=3)
plot(x=x, y=mylg, ylab="", main="")
a_values <- c(0, 2, 3)
# apply
sapply(X=a_values, FUN=function(a) {
  points(x=x,
         y=mylogistic(x=x, a=a, b=3),
         pch=a) #pch: point character
  })
```



Exercises: Vectorization

Compute the mean across rows of a matrix `M` without *for* loops. Compute the also across columns.

```
M <- matrix(data=rep(1:7, times=7) + rnorm(49),
            nrow=7)
print(M)
```

```
##            [,1]       [,2]      [,3]      [,4]     [,5]     [,6]      [,7]
## [1,] -0.4676450 0.05302214 0.5628516 0.7320667 2.229276 1.491385 1.4008347
## [2,]  2.5462929 1.59778757 1.4173676 0.9044022 3.120027 3.642906 0.9016816
## [3,]  0.7673185 3.05612025 5.3616330 1.1018916 2.924626 3.328591 3.1305792
## [4,]  4.2882842 3.84370521 3.0182713 5.4605549 4.293287 5.461393 4.7646588
## [5,]  5.5438345 3.81149423 2.8628211 3.4892120 4.971104 4.601920 3.1051437
## [6,]  4.8080152 7.28226831 5.6758617 5.4796832 5.602184 5.644404 5.7369617
## [7,]  6.9823077 6.57390567 7.5531834 5.8226666 7.655481 7.503562 6.7921217
```

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

```
## [1] 0.8573987 2.0186378 2.8101085 4.4471649 4.0550758 5.7470539 6.9833183
```

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

```
## [1] 3.495487 3.745472 3.778856 3.284354 4.399426 4.524880 3.690283
```

Note the notation: `mean`, rather than `mean()`

Tipp: Use the `rowMeans()` and `colMeans()` as shorthands.

More Exersises (Optional)

Exercises: Combine vectorization and functions

- Write a function that converts a set of numbers (i.e., a vector) into *z*-scores, i.e. number minus mean divided by standard deviation.
- Convert the matrix `M` to *z*-scores by rows

```r
mkz <- function(x) {
  return((x - mean(x)) / sd(x))
}
zmx <- t(apply(X=M, MARGIN=1, FUN=mkz))
```

Exercise: Iterative *for* loops

- Given a vector of `experiences`, compute a running mean current `belief` vector, that takes the mean of all previous experiences and store it.
- Repeat computing the running mean current `belief` with the past `5`, rather than the full past.
- (Optional) (a) Compute the running mean current `belief` such that the earlier experiences are weighted less than the later ones (forgetting curve). (b) Compute a Fibonacci sequence for the first five numbers. (c) Compute the ourcome of a random walk based on a standard normal distribution.

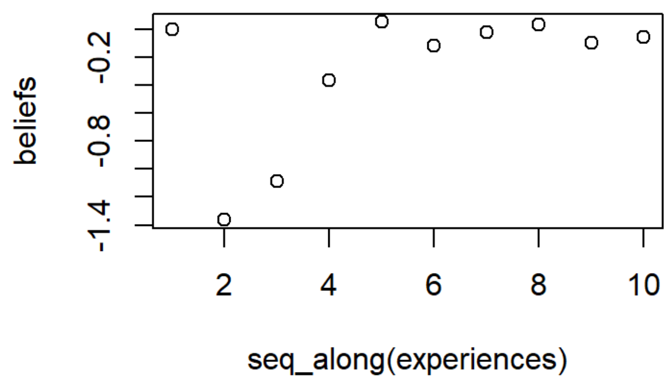
```
experiences <- rnorm(10)
print(experiences)
```

```
##  [1] -1.04187663 -1.54222061  0.75277806 -0.26536990  1.20660580 -0.79796857
##  [7]  0.09586926 -0.13050400  1.38843020 -0.25001931
```

Exercise: Iterative *for* loops

- Given a vector of `experiences`, compute a running mean current `belief` vector, that takes the mean of all previous experiences and store it. Repeat with the past `5`, rather than the full past.

```
experiences <- rnorm(10)
beliefs <- 0
for (t in 2:length(experiences)) {
  beliefs[t] <- mean(experiences[1:(t-1)])
}
plot(x=seq_along(experiences), y=beliefs)
```



Exercise: Combine *if* statements and *for* loops

Using *for* and *if*, get the difference between `predictions` and `observations` depending on a variable called `type`: if `type="absolute"`, compute the absolute difference; if `type` is `"squared"` the square the difference, and if type is `"squared-root"` the root of the square difference.

```
predictions <- c(.76, .89, .12, .34, .50)
observations <- c(0, 1, 0, 0, 1)
type <- "absolute"
```

```
y <- difference <- NULL
for (i in 1:length(predictions)) {
  difference[i] <- predictions[i] - observations[i]
  if (type == "absolute") {
    y[i] <- abs(difference[i])
  } else if (type == "squared") {
    y[i] <- difference[i]^2
    if (type == "squared-root") {
      y[i] <- sqrt(y[i])
    }
  } else {
    print("`type` must be 'absolute' or 'squared' or 'squared-root'")
  }
}
print(mean(y))
```

```
## [1] 0.366
```

Some More Exercises

Exercise: **break** to end your *for* loops

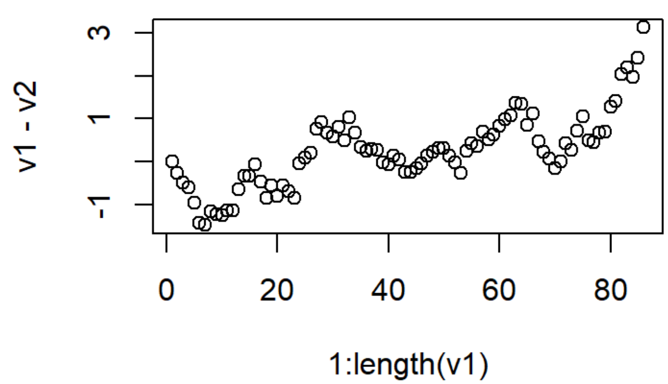
We can represent somebody gathering knowledge about two options on 365 days but stopping to gather knowledge when they believe one of the option is at least 3 better than the other.

In a *for* loop, draw from two normally distributed numbers that have *M=0* and *SD=0.20* 365 times and store the sum of these draws, but stop the process when the current draws differ by 3 or more

(Optional) Repeat for somebody who stops when the running average of the draws differs by more than 10 percent

Exercise: **break** to end your *for* loops

```r
v1 <- v2 <- 0
for (i in 2:365) {
  v1[i] <- v1[i-1] + rnorm(1, 0, 0.2)
  v2[i] <- v2[i-1] + rnorm(1, 0, 0.2)
  if (abs(v1[i] - v2[i]) > 3) {
    break
  }
}
plot(x=1:length(v1), y=v1-v2, main="")
```



Exercise: **break** to end your *for* loops

We can represent somebody who stops a process probabilistically after accumulating resources from a random process such that the stopping probability is proportional to their resources.

In a *for* loop, draw from two normally distributed numbers that have *M=0* and *SD=0.20* 365 times and compute the total resources and stop the process with a probability inversely proportional to the distance of the total accumulated resources from 1000.

(Optional) Repeat for somebody who stops proportional to when the growth in comparison to the average growth is rather a lot. Use a `while()` loop to implement the same.

Exercise: **break** to end your *for* loops

```
v1 <- 0
for (i in 2:365) {
  v1[i] <- v1[i-1] + rnorm(1, 0, 0.2)
  prob <- (1000-v1[i]) / max(1000, 1000-v1[i])
  if (rbinom(1, 1, prob=(1-prob)) == 1) {
    break
  }
}
plot(x=1:length(v1), y=v1, main="")
```