03 Programming basics

Soyoung Park

Pusan National University
Department of Statistics

Conditional expressions

The most common conditional expression is the if-else statement.

Here is a very simple example showing the general structure of an if-else statement. The basic idea is to print the reciprocal of a unless a is 0:

```
a <- 0

if(a!=0){
  print(1/a)
} else{
  print("No reciprocal for 0.")
}
#> [1] "No reciprocal for 0."
```

Conditional expressions

Let's look at one more example using the US murders data frame:

```
library(dslabs)
data(murders)
murder_rate <- murders$total / murders$population*100000
```

Conditional expressions

Here is a very simple example that tells us which states, if any, have a murder rate lower than 0.5 per 100,000. The `if` statement protects us from the case in which no state satisfies the condition.

```
ind <- which.min(murder_rate)

if(murder_rate[ind] < 0.5){
  print(murders$state[ind])
} else{
  print("No state has murder rate that low")
}
#> [1] "Vermont"
```

Conditional expressions

If we try it again with a rate of 0.25, we get a different answer:

```
if(murder_rate[ind] < 0.25){
  print(murders$state[ind])
} else{
  print("No state has a murder rate that low.")
}
#> [1] "No state has a murder rate that low."
```

A related function that is very useful is `ifelse`.

```
a <- 0
ifelse(a > 0, 1/a, NA)
#> [1] NA
```

Conditional expressions

The function is particularly useful because it works on vectors.

```r
a <- c(0, 1, 2, -4, 5)
result <- ifelse(a > 0, 1/a, NA)
```

This table helps us see what happened:

a	is_a_positive	answer1	answer2	result
0	FALSE	Inf	NA	NA
1	TRUE	1.00	NA	1.0
2	TRUE	0.50	NA	0.5
-4	FALSE	-0.25	NA	NA
5	TRUE	0.20	NA	0.2

Conditional expressions

Here is an example of how this function can be readily used to replace all the missing values in a vector with zeros:

```
data(na_example)
no_nas <- ifelse(is.na(na_example), 0, na_example)
sum(is.na(no_nas))
#> [1] 0
```

Conditional expressions

Two other useful functions are `any` and `all`. The `any` function takes a vector of logicals and returns `TRUE` if any of the entries is `TRUE`. The `all` function takes a vector of logicals and returns `TRUE` if all of the entries are `TRUE`. Here is an example:

```
z <- c(TRUE, TRUE, FALSE)
any(z)
#> [1] TRUE
all(z)
#> [1] FALSE
```

Defining functions

We can compute the average of a vector `x` using the `sum` and `length` functions: `sum(x)/length(x)`.

A simple version of a function that computes the average can be defined like this:

```
avg <- function(x){
  s <- sum(x)
  n <- length(x)
  s/n
}
```

Now `avg` is a function that computes the mean:

```
x <- 1:100
identical(mean(x), avg(x))
#> [1] TRUE
```

Defining functions

The general form of a function definition looks like this:

```
my_function <- function(VARIABLE_NAME){
  perform operations on VARIABLE_NAME and calculate VALUE
  VALUE
}
```

Defining functions

For example, we can define a function that computes either the arithmetic or geometric average depending on a user defined variable like this:

```
avg <- function(x, arithmetic = TRUE){
  n <- length(x)
  ifelse(arithmetic, sum(x)/n, prod(x)^(1/n))
}
```

We will learn more about how to create functions through experience as we face more complex tasks.

Namespaces

Once you start loading several add-on packages for some of your analysis, it is likely that two packages use the same name for two different functions.

In fact, you have already encountered this because both **dplyr** and the R-base **stats** package define a `filter` function. We know this because when we first load **dplyr** we see the following message:

```
The following objects are masked from 'package:stats':

    filter, lag

The following objects are masked from 'package:base':

    intersect, setdiff, setequal, union
```

Namespaces

These functions live in different *namespaces*. R will follow a certain order when searching for a function in these *namespaces*.

```
search()
```

The first entry in this list is the global environment which includes all the objects you define.

Namespaces

So what if we want to use the **stats** `filter` instead of the **dplyr** filter? You can force the use of a specific namespace by using double colons (**::**) like this:

```
stats::filter
```

If we want to be absolutely sure that we use the **dplyr** `filter`, we can use

```
dplyr::filter
```

Also note that if we want to use a function in a package without loading the entire package, we can use the double colon as well.

For-loops

The formula for the sum of the series $1 + 2 + \cdots + n$ is $n(n+1)/2$. What if we weren't sure that was the right function? How could we check?

Using what we learned about functions we can create one that computes the S_n:

```
compute_s_n <- function(n){
  x <- 1:n
  sum(x)
}
```

How can we compute S_n for various values of n say $n = 1, \ldots, 25$? Do we write 25 lines of code calling `compute_s_n`? No, that is what for-loops are for in programming.

For-loops

Perhaps the simplest example of a for-loop is this useless piece of code:

```
for(i in 1:5){
  print(i)
}
#> [1] 1
#> [1] 2
#> [1] 3
#> [1] 4
#> [1] 5
```

For-loops

Here is the for-loop we would write for our S_n example:

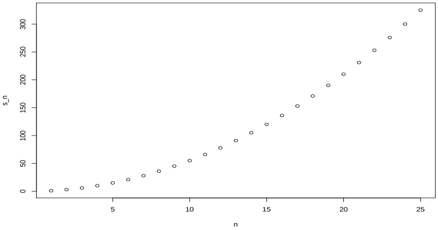
```
m <- 25
s_n <- vector(length = m) # create an empty vector
for(n in 1:m){
  s_n[n] <- compute_s_n(n)
}
```

In each iteration $n = 1, n = 2$, etc..., we compute S_n and store it in the nth entry of `s_n`.

For-loops

Now we can create a plot to search for a pattern:

```
n <- 1:m
plot(n, s_n)
```



If you noticed that it appears to be a quadratic, you are on the right track because the formula is $n(n+1)/2$.

Vectorization and functionals

A *vectorized* function is a function that will apply the same operation on each of the vectors.

```
x <- 1:10
sqrt(x)
#>  [1] 1.00 1.41 1.73 2.00 2.24 2.45 2.65 2.83 3.00 3.16
y <- 1:10
x*y
#>  [1]   1   4   9  16  25  36  49  64  81 100
```

To make this calculation, there is no need for for-loops.

Vectorization and functionals

For instance, the function we just wrote, `compute_s_n`, does not work element-wise since it is expecting a scalar. This piece of code does not run the function on each entry of `n`:

```
n <- 1:25
compute_s_n(n)
```

Functionals are functions that help us apply the same function to each entry in a vector, matrix, data frame, or list.

Vectorization and functionals

Here we cover the functional that operates on numeric, logical, and character vectors: `sapply`.

```
x <- 1:10
sapply(x, sqrt)
#>  [1] 1.00 1.41 1.73 2.00 2.24 2.45 2.65 2.83 3.00 3.16
```

The function sapply permits us to perform element-wise operations on any function. Here is how it works:

```
n <- 1:25
s_n <- sapply(n, compute_s_n)
```

Vectorization and functionals

Other functionals are `apply`, `lapply`, `tapply`, `mapply`, `vapply`, and `replicate`. We mostly use `sapply`, `apply`, and `replicate`, but we recommend familiarizing yourselves with the others as they can be very useful.

Exercises

1. What will this conditional expression return?

```
x <- c(1,2,-3,4)

if(all(x>0)){
  print("All Postives")
} else{
  print("Not all positives")
}
```

Exercises

2. Which of the following expressions is always `FALSE` when at least one entry of a logical vector `x` is TRUE?

a. `all(x)`
b. `any(x)`
c. `any(!x)`
d. `all(!x)`

Exercises

3. The function `nchar` tells you how many characters long a character vector is. Write a line of code that assigns to the object `new_names` the state abbreviation when the state name is longer than 8 characters.

4. Create a function `sum_n` that for any given value, say *n*, computes the sum of the integers from 1 to n (inclusive). Use the function to determine the sum of integers from 1 to 5,000.

5. Create a function `altman_plot` that takes two arguments, x and y, and plots the difference against the sum.

Exercises

6. After running the code below, what is the value of `x`?

```
x <- 3
my_func <- function(y){
  x <- 5
  y+5
}
```

Exercises

7. Write a function `compute_s_n` that for any given n computes the sum $S_n = 1^2 + 2^2 + \cdots + n^2$. Report the value of the sum when $n = 10$.

8. Define an empty numerical vector `s_n` of size 25 using `s_n <- vector("numeric", 25)` and store in the results of $S_1, S_2, \ldots, S_{25}$ using a for-loop.

9. Repeat exercise 8, but this time use `sapply`.

Exercises

10. Repeat exercise 8, but this time use `map_dbl`.
11. Plot S_n versus n. Use points defined by $n = 1, \ldots, 25$.
12. Confirm that the formula for this sum is $S_n = n(n+1)(2n+1)/6$.