Programming Basics

W. Evan Johnson, Ph.D.
Professor, Division of Infectious Disease
Director, Center for Data Science
Rutgers University – New Jersey Medical School

2024-05-24

Programming Basics

By coding in R, we can efficiently perform exploratory data analysis, build data analysis pipelines, and prepare data visualization to communicate results. However, R is not just a data analysis environment but a programming language.

Here we introduce three key programming concepts: **conditional expressions**, **for-loops**, and **functions**. These are not just key building blocks for advanced programming, but are sometimes useful during data analysis.

Conditional Expressions

Conditional expressions are one of the basic features of programming. They are used for what is called **flow control**.

The most common conditional expression is the **if-else** statement. In R, we can actually perform quite a bit of data analysis without conditionals. However, they do come up occasionally, and you will need them once you start writing your own functions and packages.

Conditional Expressions

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

```
x <- 0

if(x!=0){
  print(1/x)
} 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 if the state with the lowest murder has a rate lower that a user definted cutoff, e.g., say lower than 0.5 per 100,000. The `if` statement protects us from the case in which no state satisfies the condition.

```
rate_cut <- .5
ind <- which.min(murder_rate)

if(murder_rate[ind] < rate_cut){
  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:

```
rate_cut <- .25

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

```
## [1] "No state has a murder rate that low."
```

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
```

Conditional Expressions

Returning to our murders example:

```
rate_cut <- 0.71  # murder rate in Italy

if(any(murder_rate < rate_cut)){
  print(murders$state[murder_rate < rate_cut])
} else{
  print("No state has a murder rate that low.")
}
```

```
## [1] "Hawaii"         "Iowa"           "New Hampshire" "North Dakota"
## [5] "Vermont"
```

So these states have a lower murder rate than Italy.

Conditional Expressions

A related function that is very useful is `ifelse`. This function takes three arguments: a logical and two possible answers. If the logical is `TRUE`, the value in the second argument is returned and if `FALSE`, the value in the third argument is returned. Here is an example:

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

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

Conditional Expressions

The `ifelse` function is particularly useful because it works on vectors. It examines each entry of the logical vector and returns elements from the vector provided in the second argument, if the entry is `TRUE`, or elements from the vector provided in the third argument, if the entry is `FALSE`.

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

```
## [1]  NA 1.0 0.5  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
```

Defining Functions

As you become more experienced, you will find yourself needing to perform the same operations over and over.

A simple example is computing averages. We can compute the average of a vector `x` using the `sum` and `length` functions: `sum(x)/length(x)`. Because we do this repeatedly, it is much more efficient to write a function that performs this operation.

This particular operation is so common that someone already wrote the `mean` function and it is included in base R. However, you will encounter situations in which the function does not already exist, so R permits you to write your own.

Defining Functions

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
}
```

Defining Functions

Now avg is a function that computes the mean:

```
x <- 1:100
identical(mean(x), avg(x))
```

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

Defining Functions

Notice that variables defined inside a function are not saved in the workspace. So while we use `s` and `n` when we call `avg`, the values are created and changed only during the call. Here is an illustrative example:

```
s <- 3
avg(1:10)
```

```
## [1] 5.5
```

```
s
```

```
## [1] 3
```

Note how `s` is still 3 after we call `avg`.

Defining Functions

In general, functions are objects, so we assign them to variable names with <-. The call to `function` tells R you are about to define a function. The general form (pseudocode) looks like this:

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

Defining Functions

The functions you define can have multiple arguments as well as default values. 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 becoming more of an R expert user, you will likely need to load several add-on packages for some of your analysis.

Once you start doing this, it is likely that two packages use the same name for two different functions. And often these functions do completely different things.

In fact, you have already encountered this because both **dplyr** and the R-base **stats** package define a `filter` function. There are five other examples in **dplyr**. 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

So what does R do when we type `filter`? Does it use the **dplyr** function or the **stats** function? From our previous work we know it uses the **dplyr** one. But what if we want to use the **stats** version?

These functions live in different **namespaces**. R will follow a certain order when searching for a function in these **namespaces**. You can see the order by typing:

```
search()
```

```
##  [1] ".GlobalEnv"        "package:dslabs"    "package:lubridate"
##  [4] "package:forcats"   "package:stringr"   "package:dplyr"
##  [7] "package:purrr"     "package:readr"     "package:tidyr"
## [10] "package:tibble"    "package:ggplot2"   "package:tidyverse"
## [13] "package:stats"     "package:graphics"  "package:grDevices"
## [16] "package:utils"     "package:datasets"  "package:methods"
## [19] "Autoloads"         "package:base"
```

Namespaces

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

So what if we want to use the **stats** `filter` instead of the **dplyr** filter but **dplyr** appears first in the search list? 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
```

Namespaces

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 more on this more advanced topic we recommend the R packages book[1].

[1] http://r-pkgs.had.co.nz/namespace.html

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)
}
```

For-loops

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. In this case, we are performing exactly the same task over and over, and the only thing that is changing is the value of n.

For-loops let us define the range that our variable takes (in our example $n = 1, \ldots, 10$), then change the value and evaluate expression as you **loop**.

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:

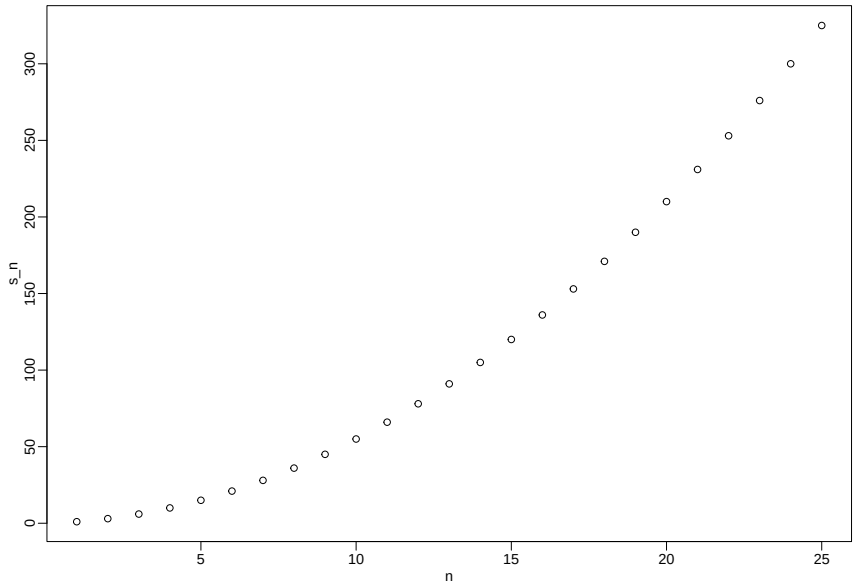
```
m <- 25
s_n <- vector(length = m) # create an empty vector

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

Although for-loops are an important concept to understand, in R we rarely use them. As you learn more R, you will realize that **vectorization** is preferred over for-loops since it results in shorter and clearer code.

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] 1.000000 1.414214 1.732051 2.000000 2.236068 2.4494
##  [9] 3.000000 3.162278
```

```
y <- 1:10
x*y
```

```
##  [1]   1   4   9  16  25  36  49  64  81 100
```

Vectorization and Functionals

To make this calculation, there is no need for for-loops. However, not all functions work this way. 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)
```

Vectorization and Functionals

Functionals are functions that help us apply the same function to each entry in a vector, matrix, data frame, or list. Here we cover the functional that operates on numeric, logical, and character vectors: `sapply`.

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

```
x <- 1:10
sapply(x, sqrt)
```

```
##  [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.4494
##  [9] 3.000000 3.162278
```

Vectorization and Functionals

In `sapply`, each element of `x` is passed on to the function `sqrt` and the result is returned. These results are concatenated. In this case, the result is a vector of the same length as the original `x`. This implies that the for-loop above can be written as follows:

```
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` in this book, but we recommend familiarizing yourselves with the others as they can be very useful.

Exercises

Now open the **Programming Basics Exercises** file and complete Exercises 1-12.

Session Info

```
sessionInfo()
```

```
## R version 4.4.0 (2024-04-24)
## Platform: aarch64-apple-darwin20
## Running under: macOS Sonoma 14.2.1
##
## Matrix products: default
## BLAS:   /Library/Frameworks/R.framework/Versions/4.4-arm64/Resources/lib/libRblas.0.dylib
## LAPACK: /Library/Frameworks/R.framework/Versions/4.4-arm64/Resources/lib/libRlapack.dylib;  LAPACK vers
##
## locale:
## [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
##
## time zone: America/Denver
## tzcode source: internal
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
##  [1] dslabs_0.8.0    lubridate_1.9.3 forcats_1.0.0   stringr_1.5.1
##  [5] dplyr_1.1.4     purrr_1.0.2     readr_2.1.5     tidyr_1.3.1
##  [9] tibble_3.2.1    ggplot2_3.5.1   tidyverse_2.0.0
##
## loaded via a namespace (and not attached):
##  [1] gtable_0.3.5       compiler_4.4.0     tinytex_0.50       tidyselect_1.2.1
##  [5] scales_1.3.0       yaml_2.3.8         fastmap_1.1.1      R6_2.5.1
##  [9] generics_0.1.3     knitr_1.46         munsell_0.5.1      RColorBrewer_1.1-3
## [13] pillar_1.9.0       tzdb_0.4.0         rlang_1.1.3        utf8_1.2.4
## [17] stringi_1.8.3      xfun_0.43          timechange_0.3.0   cli_3.6.2
## [21] withr_3.0.0        magrittr_2.0.3     digest_0.6.35      grid_4.4.0
## [25] rstudioapi_0.16.0  hms_1.1.3          lifecycle_1.0.4    rafalib_1.0.0
## [29] vctrs_0.6.5        evaluate_0.23      glue_1.7.0         fansi_1.0.6
## [33] colorspace_2.1-0   rmarkdown_2.26     tools_4.4.0        pkgconfig_2.0.3
## [37] htmltools_0.5.8.1
```