Introduction to Coding in R

EKB

2024-02-01

Introduction to R and RStudio

Student Learning Outcomes

- Students will be able to explain the benefit of each of the four main panels in RStudio
- Students will be able to do the following in the R languages:
  - perform mathematics
  - assign objects
  - use functions
  - explore and describe vectors (1D data)

RStudio Cloud Tour

Let's first explore what each of the panels in RStudio do.

1. *Source* (upper left): This is where documents which have data or code in them are opened. You can save all the code you type here for future (re)use, which is a big reason coding in R is reproducible.
2. *Console* (bottom left): This is where code from the source is "run" and you see the outputs. You can also execute lines of code which you type into the console, but they will not be saved. You can think of this section as where RStudio really interfaces with R–it is where R actually evaluates code within RStudio.
3. *Environment* (upper right): This panel becomes more helpful as you get familiar with R and RStudio. It keeps track of data objects and other items you have created and gives a bit of information about them.
4. *Files/Help/etc.* (bottom right): This panel is (clearly) very multifaceted. The Files tab lets you see all the files in your current workspace. For us, the Help tab is probably what we will use the most. This is where we can search the R documentation for information about functions we use.

Basics of Coding in R

How do you "run" code? Running or executing code means that you are sending a line of code to the console for R to interpret it.

In the source, there are a few different ways to run code.

- Sometimes you want to run only one line of code at a time. You can do this by putting your cursor on the line you want to run and either hitting the "Run" button in the upper right-hand corner of the source panel or holding down `Ctrl` + `Enter` (`Cmd` + `Enter` on Macs). If you want to run a couple lines of code, you can highlight them and do the same thing

- If you want to run an entire code chunk (see below), you can click on the green arrow on the right side of the code chunk.

In the console, you hit `Enter`.

Using R as a calculator

For example:

```
3 + 5
```

```
## [1] 8
```

```
15 / 5
```

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

```
4^2
```

```
## [1] 16
```

Let's practice in the console, too.

You can do basic math in the console. The console only understands R code, so we don't need to use the `{r}` notation; we can type numbers and mathematical symbols. Try multiplying 5 and 3 (hint: * means multiply) in the console.

Assigning Objects

Assignments are really key to almost everything we do in R. This is how we create permanence in R. Anything can be saved to an object, and we do this with the assignment operator, `<-`.

The short-cut for `<-` is `Alt + -` (or `Option + -` on a Mac). We will be typing this *a lot*.

```
height <- 47.5
age <- 122
```

We can also perform mathematical functions on numeric objects.

```
# we can add comments to our code by using the # sign
# R doesn't read anything past a #, so you can either put one at the beginning of a whole line or after

height <- height * 2        # multiply
age <- age - 20         # subtract
height_index <- height/age  # divide
```

Functions

Functions are pre-written bits of codes that perform specific tasks for us.

Functions are always followed by parentheses. Anything you type into the parentheses are called arguments. Arguments allow us to give the function additional information about how we want it to perform its task.

```r
sqrt(10)
```

```
## [1] 3.162278
```

```r
weight_kg <- sqrt(10) # square root

round(weight_kg) # rounding
```

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

```r
round(weight_kg, digits = 2) # round to 2 digits past 0
```

```
## [1] 3.16
```

To get more information about a function, use the help function.

```r
# these two lines of code do the same thing
help(mean)
?mean
```

Let's Practice!

Write some code that calculates the sum of the `weight_kg` object and the `height` object. You'll want to use the `sum()` function. Save the result as an object called `weight_and_height`.

```r
weight_and_height <- sum(weight_kg, height)
```

Vectors

Vectors are the most common and basic data type in R. They make up most of the other data types we will work with in R. They are composed of series of values, which can either be numbers or characters or other types of data.

We use the `c()` function (stands for combine) to create a vector.

```r
# Let's create a vector of animal weights (numeric)
weight_g <- c(60, 65, 82)
weight_g
```

```
## [1] 60 65 82
```

```r
# A vector can also contain character strings (character)
animals <- c("mouse", "rat", "penguin", "rat")
animals
```

```
## [1] "mouse"   "rat"     "penguin" "rat"
```

```r
# Logical vector
mammal <- c(TRUE, TRUE, FALSE, TRUE)
mammal
```

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

There are many functions we can use to look at vectors and learn more about them.

```r
# how many elements
length(weight_g)
```

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

```r
length(animals)
```

```
## [1] 4
```

```r
# type of data we are working with
class(weight_g)
```

```
## [1] "numeric"
```

```r
class(animals)
```

```
## [1] "character"
```

```r
# structure of an object
str(weight_g)
```

```
##  num [1:3] 60 65 82
```

```r
str(animals)
```

```
##  chr [1:4] "mouse" "rat" "penguin" "rat"
```

```r
# find the unique values
unique(animals)
```

```
## [1] "mouse"   "rat"     "penguin"
```

Vectors can only be one data type. Let's experiment with that.

```
test_vec <- c(weight_g, animals, mammal)
test_vec
```

```
##  [1] "60"      "65"      "82"      "mouse"   "rat"     "penguin" "rat"
##  [8] "TRUE"    "TRUE"    "FALSE"   "TRUE"
```

```
class(test_vec) # coerced everything into character (don't know how to make words numeric)
```

```
## [1] "character"
```

Sub-setting by Index

Sometimes we want to pull out and work with specific values from a vector. This is called sub-setting (taking a smaller set of the original).

One way to do this is by an "index," meaning the position of the value or object in the vector. To do this, we use square brackets and a number to indicate the position.

```
weight_g[2]
```

```
## [1] 65
```

```
weight_g[c(2,4)]
```

```
## [1] 65 NA
```

```
weight_g[c(1:4)]
```

```
## [1] 60 65 82 NA
```

Conditional subsetting

Another way in which we can subset data is through conditions. The vector will only return data which meets the conditions we set.

```
weight_g > 60
```

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

```
weight_g[weight_g > 55]
```

```
## [1] 60 65 82
```

```
animals == "rat"
```

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

```r
animals[animals == "rat"]
```

```
## [1] "rat" "rat"
```

Group Challenge

Let's practice! Write a few lines of code that do the following:

- create a vector with even numbers from 1 to 10 (including 10)
- assign the vector to an object named `vec`
- using the *index* method, subset `vec` to include the last 3 numbers (should include 6, 8, 10)
- find the sum of the numbers (hint: use the `sum()` function)

```r
# the answer you should get out is 24
vec <- c(2, 4, 6, 8, 10)
vec
```

```
## [1]  2  4  6  8 10
```

```r
last3 <- vec[3:5]
last3
```

```
## [1]  6  8 10
```

```r
sum(last3)
```

```
## [1] 24
```

If you've finished that task, try to get the same result using *conditional* sub-setting.

```r
sum(vec[vec > 5])
```

```
## [1] 24
```