CS100: Studio 4

Programming Practice

October 6, 2021

Instructions

During today’s studio, you will be practicing with some programming fundamentals. Please write all of your code in an R script (not in an R markdown file, like usual).

Upon completion of all tasks, a TA will give you credit for today’s studio. If you do not manage to complete all the assigned work during the studio period, do not worry. You can continue to work on this assignment until Tuesday, October 12, at 10 PM. Come by TA hours any time before then to show us your completed work and get credit for today’s studio.

Objectives

By the end of this studio, you will be able to:

- program with for conditionals
- program with for loops
- program with functions

Part 1: Programming Review in R

The first task in this studio is a very simple programming task, designed to help you synthesize the programming concepts (if statements, for loops, and functions) you’ve been learning about recently. Each of these concepts is described in this write-up, but you are free to skip ahead directly to the task, if you feel confident in your knowledge of these concepts.

If Statements

If statements are used to direct the flow of a program based on whether or not various conditions hold. In computer science parlance, these conditions are called predicates.

Below are some examples of predicates. Think about whether each one is true or false:

TRUE

# == tests equality
1 == 0

# != tests inequality
1 != 0

100 > 1000

x <- "erin"
y <- "anna"
x < y
You can combine predicates using \&\& (and) and || (or). Predicates connected by \&\& are true if both sub-predicates are true, and predicates connected by || are true if either (or both) sub-predicates are true. You can also negate a predicate using ! (bang) so that true predicates become false, and vice versa.

Here are some more examples. Again, think about whether each one is true or false:

\begin{verbatim}
! FALSE
TRUE \&\& FALSE
TRUE || FALSE
3 * 4 == 12
! (3 * 4 != 12)
5 == 5 \&\& 7 > 6
5 == 5 \&\& ! (7 > 6)
! (5 == 5 \&\& 7 > 6)
(1 > 2 || 4 > 3) || (10 >= 1 \&\& 0 != 1)
\end{verbatim}

Now that you understand predicates, you can use them as conditions in if statements:

\begin{verbatim}
if (TRUE) {
  print("This will print!")
}

if (5 > 10) {
  print("This won’t print!")
}
\end{verbatim}

If you want to check a chain of predicates, you can make use of the else and else if statements.

\begin{verbatim}
x <- 5

# The \%\% operator returns the remainder of two numbers. It’s called the mod (short for modulo) operator!
if (x \%\% 3 == 0) {
  print("This will print if x is a multiple of 3")
} else {
  print("This will print in all other cases, namely when x is not a multiple of 3")
}

if (x == 5) {
  print("x equals 5")
} else if (x == 6) {
  print("x equals 6")
} else if (x == 7) {
  print("x equals 7")
} else if (x >= 8) {
  print("x is greater than or equal to 8")
} else {
  print("x is less than 5")
}
\end{verbatim}

**For Loops** Often in programming, you will find yourself wanting to perform an operation more than once. Iterating over blocks of code, rather than repeating yourself, is not only convenient, it helps prevent bugs. A for loop is a
handy iterator. Below is an example:

```r
for (i in 1:10) {
  print("Hello")
}
```

This `for` loop will print “Hello” 10 times. The way it works is: the variable `i` changes value each time the loop is entered. The first time its value is 1, next it is 2, then 3, and so on, all the way up to 10, since the range of values specified is `1:10`.

Since `i` is a variable, the code inside a `for` loop can refer to `i`, and take advantage of the fact that its value is updated with each iteration. Here is another example, in which the numbers 10 through 20 (inclusive), are printed out:

```r
for (i in 10:20) {
  print(i)
}
```

It is also possible to do the reverse, namely print 20 through 10 instead:

```r
for (i in 20:10) {
  print(i)
}
```

Note, however, that the output of this program is identical to that of our first example:

```r
for (i in 20:10) {
  print("Hello")
}
```

**Programming Tip:** Pick particular variable names (i is a popular one, and so is j) to use as counters in your loops, and do not use these variables elsewhere in your programs. That way, it will be easy to keep track of your counters’ values, and you won’t get confused by their values changing unexpectedly.

---

**Functions**  We’ve talked a lot about using built-in functions, but a major part of programming is writing your own functions! The most readable (and hence, bug-free) code usually consists of lots and lots of small functions pieced together into one large program.

This is what a typical function looks like:

```r
defname.of.function <- function(argument1, argument2) {
  statements
  something_to_return
}
```

Observe that it has a name (i.e., `name.of.function`), and that it has a body. The body is the code enclosed within curly brackets `{ and }`.

Most functions take as input at least one argument, although none are required. This example takes two. Finally, a function usually returns a value to its caller, although it doesn’t have to. (Programming languages are very flexible!)

In R, the value that is calculated on the last line of a function is automatically returned to the function’s caller (regardless of whether or not the caller might have any use for that value).

Note that you can also explicitly return values using the `return` keyword, as follows:

```r
defname.of.function <- function(argument1, argument2) {
  statements
  return(something_to_return)
}
```

Here’s an example of an extremely simple function that adds one to its argument, which is presumably a number:

```r
defadd.one <- function(num) {
  added_one <- num + 1
```
As you can see, this function creates an object, \texttt{added_one}, which is then returned in the last line of the function. For example, the value of \texttt{add.one(5)} is 6.

Here’s another slightly cleaner way we could have written \texttt{add.one}:

\begin{verbatim}
add.one <- function(num) {
  num + 1
}
\end{verbatim}

Let’s look at another example. This function that takes as input two arguments and returns their sum:

\begin{verbatim}
add.two.num <- function(x, y) {
  x + y
}
\end{verbatim}

For example, \texttt{add.two.num(1, 2)} evaluates to 3. Here’s a variant of the above, where we assign a default value of 1 to the second argument.

\begin{verbatim}
add.two.num <- function(x, y = 1) {
  x + y
}
\end{verbatim}

Now \texttt{add.two.num} behaves just like \texttt{add.one}, adding one to its first argument when the second is omitted, and summing its arguments when two values are given.

More details about functions in R can be found here.

\textbf{Task} \hspace*{1em} Now that you’ve learned about if-statements, for-loops, and functions, write a function \texttt{sum.evens} that takes as input a number \texttt{n} and returns the sum of all even numbers between 1 and \texttt{n}, inclusive.

\textit{Hint:} Use the \texttt{mod} operator, \texttt{%%} in R, to test whether a number is even or odd.

\textit{More detailed hint:} Modular arithmetic gives the remainder when dividing some number \texttt{x} by some number \texttt{y}. For example, \texttt{7 \texttt{%%} 3}, read as “7 mod 3”, gives 1 because 7 divided by 3 is 2 with a remainder of 1. Solve the following examples by hand, and then run them in R to verify your understanding.

\begin{verbatim}
8 \texttt{%%} 6
7 \texttt{%%} 4
10 \texttt{%%} 5
3 \texttt{%%} 3
\end{verbatim}

\textbf{Part 2: FizzBuzz}

The FizzBuzz problem is a short programming task, often asked during software engineering interviews. Here is one variant:

Write a function called FizzBuzz that takes as input a number, and prints “Fizz!” if the number is a multiple of 3, “Buzz!” if the number is a multiple of 5, and “FIZZ BUZZ!!!” if the number is a multiple of both 3 and 5. If the number is neither a multiple of 3 nor a multiple of 5, it should just print a sad face.

Here are some sample inputs and their corresponding outputs:

\begin{verbatim}
2: :(
3: Fizz!
5: Buzz!
6: Fizz!
15: FIZZ BUZZ!!!
\end{verbatim}
Task  Write a function `fizzbuzz` that takes as input a number `n` and prints out the number as well as `Fizz!`, `Buzz!`, `FIZZ BUZZ!!!`, or `:(`, as required. Note that printing both `Fizz!` and `Buzz!` if you encounter a multiple of both 3 and 5 instead of `FIZZ BUZZ!!!` is incorrect.

Hint: You may find if else statements helpful!

Another hint: Feel free to use the `cat` function, which prints a combination of variables and strings, with spaces in between them. For example,

```r
hello <- "Hello"
world <- "world"
year <- 2021
cat(hello, world, year, "!")
```

This code will display `Hello world 2021 !`.

Once you have a working version of `FizzBuzz`, write a `for` loop that calls your `fizzbuzz` function on a vector comprising your favorite numbers.

Once your `for` loop is working, please call over a TA to review your work. This is an important checkpoint; we want to make sure that you understand these programming concepts, because you will need them for future assignments.

End of Studio

When you are done please call over a TA to review your work, and check you off for this studio. If you do not finish within the two hour studio period, remember to come to TA office hours to get checked off.