R: Atomic Vectors and Functions

Signal Data Science

We'll start with a discussion of the type system in R. Afterward, we'll move into a discussion of atomic vectors and what you can do with them. Finally, we'll cover conditional statements and custom functions.

Types

The concept of *data types* is fundamental to programming.[1] Every "basic unit" of data—every variable in R—has a particular *type*, which tells the processor what you can or can't do with it.

To determine the type of a variable, you can use `typeof()`. Although `typeof()` can in fact return any of 24 different types, in practice we'll only concern ourselves with a couple of them.[2] In particular, we'll begin by focusing on the "integer", "double" (floating-point number),[3] "character" (string), and "logical" (boolean) types.

The first two of those allow you to do simple arithmetic in R, *e.g.* you can type in `1+2` into the console and get `3` back. Let's try testing the type of a floating-point number:

```
> typeof(5.5)
[1] "double"
```

- First, pick an integer and try running `typeof()` on it. Next, try running `5.5L` and `typeof(5.5L)`. Observe both the *output* and the *warning message* of both commands. Can you figure out how to pass in an integer into `typeof()` so that it returns `"integer"` for the type?[4]

In practice, you don't actually care about the difference between how R handles integers and how R handles doubles. In a certain precise sense, "numeric"

[1] See Wikipedia: Data type.

[2] The R language specification has a list of possible types.

[3] In computer-science parlance, "double" refers to the double-precision floating-point format.

[4] R will automatically assume that a number is supposed to be a double. To make an integer, you have to explicitly specify it as an integer literal by appending `L` to the end, like with `typeof(3L)` (which has output `"integer"`). The `L` stands for for a "long" integer, which is common programming terminology for a 32-bit integer data type.

objects are a generalized structure which encapsulates all sorts of real numbers (integers, doubles, and more). What we really want to think about are numerics, characters, and logicals.[5]

- We can test if something is numeric with `is.numeric()`. Verify that `3`, `3.1`, and `5L` are all numeric.

All strings have type "character" and are defined by wrapping text in between single or double quotes (there's no difference). For example, we have `typeof("qwerty") = typeof('qwerty') = "character"`.

Finally, the boolean values `TRUE` and `FALSE` have type "logical."

- Try running the following commands: `TRUE`, `FALSE`, `T`, `F`. Think about the following commands and predict their outcome: `TRUE == FALSE`, `TRUE & (T | FALSE)`. Next, run them and check your predictions. Finally, explain the output of `F = T; F & TRUE`.[6]

In your code, *always* use `TRUE` and `FALSE` instead of `T` and `F` to avoid potential confusion.

Atomic vectors

Atomic vectors in R are analogous to lists in Python, with the exception that *they can only contain a single type*. Vectors are formed with `c()`, which stands for *combine*. For example:

```
> c(1,2)
[1] 1 2
> c(1.1, 4.5, 2.7)
[1] 1.1 4.5 2.7
> c('test', 'test2')
[1] "test"  "test2"
```

You should not think of vectors as being some sort of *enclosing structure* with values within it. Rather, the vector structure is intrinsically associated with every value, such that a single value (say, `4` or `"test"`) are themselves vectors of length 1.

- How can you construct an empty vector?

[5] `mode()` will give you the *mode* of an object as described in The New S Language. Integers and doubles both have the "numeric" mode. If you're wondering how S is relevant to R, it's because R is one of the modern *implementations* of the S language specification. It's open-source and is free software, whereas S-PLUS, the only other modern implementation of S, is proprietary.

[6] You'll notice that `T` and `F` aren't equivalent to the primitive boolean values `TRUE` and `FALSE` but rather system-predefined variables initialized to `T = TRUE` and `F = FALSE`. That's why you can set `T = F` to set `T` to be equal to `FALSE`, and even change them to non-logical values altogether such as with `T = 2^4 + 1`.

- Is there any distinction between an atomic vector and a single value of the same type? There's a function you can use to check whether or not something is an atomic vector: look back to previous exercises and try to figure out what it is.[7]
- Determine how vectors behave when you nest them inside each other.
- If you have an atomic vector `v = c(1, 2, 3)`, how can you append the value `4` to the end, forming `c(1, 2, 3, 4)`?
- Recall that vectors can only be associated with a single type. What happens when you try to add an incompatible type to a vector?
- Before doing anything, consider the three primary types which we'll be handling: numeric, logical, and character. Which is the most specific and which is the most general? Based on that, what do you expect will happen when we add an incompatible type to a vector?
- Look at the output of `c(5, FALSE, TRUE, 10)`, `c('asdf', FALSE)`, and `c(TRUE, 'qwerty', 10.10)`. What is the underlying principle behind vector type coercion?[8]

Of course, you can also explicitly coerce vectors into different types with functions like `as.character()` and `as.numeric()`. (Try this out!) However, in general, coercing a more general type to a more specific type will result in an error.[9] In particular, you will introduce `NA` values into a vector when type coercion is not possible:

```
> as.logical(c('TRUE', 'str1', 'str2'))
[1] TRUE   NA   NA
```

The `NA` is a special value that automatically takes on the type of the enclosing vector as needed. By default, `NA` is of type "logical", but you can access other `NA` types with `NA_real_` and `NA_character_`.

- Using `sum()` and `mean()`, how can you easily calculate the number and proportion of `TRUE` values in a logical vector? With *only* `mean()` and `length()`, how can you calculate the number of `TRUE`s?
- Test your knowledge of vector coercion rules by predicting the output of the following uses of `c()`:

  ```
  c(1, FALSE)
  c("a", 1)
  ```

[7] Similar to `is.numeric()`, we can use `is.atomic()` to check if something is an atomic vector. You'll find that single values also count as being atomic vectors.

[8] Vectors are automatically converted to the most general type necessary to accommodate all of the information inside. In order of increasing generality, we have the types logical, numeric, and character (with integer < double within the numeric category).

[9] In specific cases, conversion from more to less general works, *e.g.*, `as.logical('TRUE')` or `as.numeric('100')`.

```
c(list(1), "a")
c(TRUE, 1L)
```

- Why does `1 == "1"` evaluate to `TRUE`? Why does `-1 < FALSE` evaluate to `TRUE`?
- Why is the default missing value, `NA`, a logical vector? What's special about logical vectors? (Hint: think about `c(FALSE, NA_character_)`.)[10]

Conditionals and loops

Conditionals are very helpful in structuring the logic of your code. The following code example illustrates how they work:

```
x = 1
if (x > 0) {
    print("x is positive")
} else if (x < 0) {
    print("x is negative")
} else {
    print("x is zero")
}
```

Now, for practice:

- Change the values of `x` and see how the output of this code varies. Next, change this code so that it distinguishes if `x` is greater than, lesser than, or equal to 10.
- Run the following code:

  ```
  x = runif(1)
  if (x < 0.5) {
    print(x)
  }
  else {
    print('big')
  }
  ```

  Determine the source of the error and fix it. What does `runif()` do? Try to guess before looking it up in the documentation. (*Hint:* It's `r unif`, not `run if`.)

Next, read about how to use `for` loops in R.

[10] You don't want `NA`s to coerce other values in a vector to more general types; rather, you want the `NA` to be the most *specific* type, a logical, so that the presence of missing values doesn't coerce the rest of your data into an unexpected type.

- Use `paste()` to make a vector of length 30 that looks like `("label 1", "label 2", ..., "label 30")`.
- `rnorm()` is used to sample from a normal distribution. Write code that generates 10 random samples from the normal distribution, loops through them, and for each value `x` prints `x` if `x` is less than 0.5 or prints `"big"` if not.
- Calculate the sums $\sum_{i=10}^{100}\left(i^3+4i^2\right)$ and $\sum_{i=1}^{25}\left(\frac{2^i}{i}+\frac{3^i}{i^2}\right)$ using `sum()`.
- Read about how to time function calls in R. Set `x = 1:10` and play around with the different ways in which you can access parts of `x`, like with `x[1:10]`, `x[2:3]`, etc. Using `tail()`, `length()`, and `seq()`, come up with as many different ways as you can to access the last 5 elements of `x`, and time them to figure out which ones are the fastest. Afterward, check your results against the results of others.
- Create a vector of the values of $e^x\cos(x)$ at $x=3,3.1,3.2,\ldots,6$.
- Using two nested `for` loops, print out all pairs of integers from 1 to 20 without repeats (order doesn't matter). *E.g.*, don't print out both `(1,2)` and `(2,1)`.

Writing custom functions

In the following section, you'll be doing a bit of [experimental mathematics](https://en.wikipedia.org/wiki/Experimental_mathematics[11] to learn about using and writing simple functions in R.

- Read about how to define functions in R.

We'll first use custom functions to look at the unproven Collatz conjecture, proposed in 1937 by the German mathematician Lothar Collatz. Although the problem is easy to state, it has been notoriously difficult to prove. The conjecture has deep connections to the structure of the natural numbers and their prime factorizations.

- Make a function `collatz(n)` that takes in a positive integer `n` and returns `n/2` if `n` is even or `3n+1` if `n` is odd. Play around with this—what's the limiting behavior as you apply this function repeatedly, e.g. taking `collatz(collatz(collatz(collatz(collatz(n)))))`?
- For the first 100 integers, calculate the number of iterations of `collatz()` required before the output reaches 1 and use `hist()` to make a histogram of these results. (For the first part, you might find while loops useful.)

[11] V.I. Arnold was an ardent supporter and practitioner of experimental mathematics, with a book on the subject posthumously published in 2015.

Next, let's verify the identity of the Hardy–Ramanujan number, the most well known of the taxicab numbers, with some computations.

- Write a function that takes in a positive integer `n` and calculates the number of ways in which `n` is expressible as the sum of two cubes of positive integers. What is the smallest integer expressible as the sum of two cubes in two different ways?

Computational techniques are also very useful for studying the properties of Fibonacci numbers. Playing around with numerical examples can often shed light on the intuition behind relatively abstract mathematical concepts.

- Write a function `fib(n)` that returns the `n`th Fibonacci number, with `fib(1) == fib(2) == 1`. Then write a different function `fib_test()` that takes in two parameters, `n` and `k`, and for the first `n` Fibonacci numbers calculates whether or not they're divisible by `k`. (Think about what this function should return!) Play around and see if you can find any patterns (hint: try `k = 3`).

In general, instead of recalculating the output of the same subroutine repeatedly, you can store the results of computations the first time you do them and then reuse the precomputed results if you need them again. (This is called memoization, related to the method of dynamic programming.)

- Use memoization to speed up some of the code you've just written and quantify the improvements in runtime using the `tictoc` package. In particular, improve `fib_test()` and `collatz()`.