Stats 20

Discussion, Week 2

Jake Elmstedt

10/11/2020

Contents

Everything is a Vector!

When I say everything in R is a vector, what I mean is the vector is the smallest atomic building block in the R language – there's nothing smaller. That might seem strange, you may even be thinking, "Aha! What about scalar values like `1` and `2`?" Nope. Vectors. They just happen to be vectors of length-one! So, with that in mind, what would you say if I were to ask you how many vectors are created in this line of code?

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

If you said "one," BOO! You're not paying attention and we've only just begun! If you paid attention and thought about it for a moment and came up with "four," great job! You're also wrong! The *technically* correct[1] answer is five. Each scalar is a vector so that's three, then we combine them into one, length-three vector on the right hand side, *then* we copy that vector into an object `x` which is, of course, a vector making five total vector creations. Do you *ever* need to think or worry about this? No. I'm just trying to drive home the point that *everything* in R is a vector!

Creating Vectors

So, how do we create vectors? As we've seen it's pretty much impossible to *not* create a vector with just about everything you do, so the better question might be to ask "what is the *best* way to create vectors?"

There are three ways to create vectors which you will likely use more than any others.

1. Functions which build vectors
2. Vector class constructors
3. Combining vectors into new vectors

Functions

At this point in your education you've already been introduced to a couple of workhorse functions for building vectors. `seq()` and `rep()` as well as one operator`:`.

`seq()` `seq()` generates regular sequences of numbers. There are a few different ways you'll use `seq()` over and over in this class and beyond so it will be valuable to look at these common cases.

1. A regular sequence of a certain length. If you want a sequence starting at `1` and going to a number `n` there are a few ways you can accomplish this:

```
n <- 3
seq(n)
```

```
## [1] 1 2 3
```

```
seq(from = 1, to = n)
```

```
## [1] 1 2 3
```

[1] which I think everyone will agree is the best kind of correct

```r
1:n # this is a shorthand form of seq(from = 1, to = n)
```

```
## [1] 1 2 3
```

99.9% of the time though this *is not* what you want! When planning your code you should be very careful and precise about how you describe the objects you want to create!

The *vast* majority of the time when you are imagining making a vector that looks like `1, 2, 3, ... n`, what you *really* want is an increasing vector which starts at `1` and *has a length of* ***`n`***!

You can do this a couple of ways:

```r
n <- 3
seq(length.out = n)
```

```
## [1] 1 2 3
```

```r
seq_len(n) # shorthand form of seq(length.out = n)
```

```
## [1] 1 2 3
```

Why does this matter? Most of the time won't matter at all, but when it does it matters *a lot*. Imagine if, for some reason, `n = 0`. Then these produce very different results,

```r
n <- 0
1:n
```

```
## [1] 1 0
```

```r
seq_len(n)
```

```
## integer(0)
```

`seq(from, to)` and the shorthand `from:to` cannot create length-zero vectors, which again doesn't matter until it does. I promise you, it will be far easier for you in the end if you just get in the habit now of doing this. If you don't there will almost certainly be a time in the future where you are harshly reminded of this advice after spending several hours trying to debug your code.

```r
set.seed(143)
x <- sample(20, 5)
```

2. A vector containing all (or almost all) of the indices of another vector Say you have a vector `x` that look like this:

```
## [1]  6  8  1 12 20
```

If you want a vector of the indices of `x` you might try:

```
1:length(x) # NOOOOOO!
```

```
## [1] 1 2 3 4 5
```

Don't do this! For the same reasons as above, if `x` is of length-zero you'll often run into logic issues with your code. So, what should you do instead?

```
seq(length.out = length(x))
```

```
## [1] 1 2 3 4 5
```

```
seq_along(x) # shorthand for seq(length.out = length(x))
```

```
## [1] 1 2 3 4 5
```

Also seen is:

```
seq_len(length(x))
```

```
## [1] 1 2 3 4 5
```

but you should just use `seq_along(x)`.

Now, I mentioned you might want *almost all* of the indices of a vector `x`. I'm going to share with you two common needs and what I consider to be the best way to construct them.

a. All of the indices *except* the first To make a vector with all of the indices of a vector except the first I recommend using (**NOTE:** We'll talk about subsetting, particularly negative subsetting, in a moment),

```
seq_along(x)[-1]
```

```
## [1] 2 3 4 5
```

This does just what is described, it makes a vector of all of the indices of `x` then removes the first index.

So, why don't we do what many beginning (and far too many experienced) programmers do and write this?

```
seq(from = 2, to = length(x))
```

```
## [1] 2 3 4 5
```

Again, with everything you write, you should consider what can go wrong. What if `x` is length-zero (or even length-one)?

```
x <- integer(0)
seq(from = 2, to = length(x))
```

```
## [1] 2 1 0
```

```
x <- integer(1)
seq(from = 2, to = length(x))
```

```
## [1] 2 1
```

Neither of these is what we want! If we think about it carefully what do we want when `x` is length-zero?

Well, there aren't *any* indices in `x`, so if we get rid of the last index in a set of nothing we still have nothing.

```
x <- integer(0)
seq_along(x)[-1]
```

```
## integer(0)
```

And if `x` is length-one? Then it only has one index value (`1`), and if we get rid of it we should have an empty vector again...

```
x <- integer(1)
seq_along(x)[-1]
```

```
## integer(0)
```

Great!

So, when you find yourself thinking something like "I need a sequence from `2` to the end of `x`," you should catch yourself and reframe it as "I need a containing all but the first index of `x`," and write your code accordingly.

b. All of the indices *except* the last

This is the flip-side to the above and is considered a best practice for the same reason.

Instead of writing something like `seq(length(x) - 1)` and especially `1:(length(x) - 1)` it is infinitely better to write,

```
x
```

```
## [1]  1  8  9 18 11
```

```
seq_along(x[-1])
```

```
## [1] 1 2 3 4
```

Notice that we're removing the *first* element of `x` not the *last*! While you certainly *could* remove the last element, it would involve a bit more (admittedly fairly trivial) code, but there's no reason for it. Creating a vector with all of the indices of `x` except the last is an identical problem to creating a vector of all of the indices of a vector which has the same length as `x` with one of it's values removed. it doesn't matter which, so for ease we can just do the first.

`rep()` `rep()` allows you to repeat the elements of a vector, creating a new vector.

There are three different arguments you can use to control the behavior of 'rep()".

1. `times` – How many times to repeat the vector back to back.
2. `each` – How many in-place copies of each element to make.
3. `length.out` – How long to make the result vector when repeating as in #1.

We'll use the simple vector `x <- 1:3` as we explore each of these.

`times` Note that the `times` argument must be one of: * a length-one vector * a vector the same length as `x`

```
x <- 1:3
rep(x, times = 2)
```

```
## [1] 1 2 3 1 2 3
```

```
rep(x, times = c(2, 1, 3))
```

```
## [1] 1 1 2 3 3 3
```

```
rep(x, times = c(1, 3))
```

```
## Error in rep(x, times = c(1, 3)): invalid 'times' argument
```

NOTE: When `times` is a vector the same length as `x`, `rep()` performs the repetitions in a manner similar to how it does when you use the `each` argument. This is unintuitive and the results you get from `rep(x, times = c(2, 1, 3))` should (arguably) be the result from `rep(x, each = c(2, 1, 3))` (which is an invalid command).

`each` The `each` argument in `rep()` accepts a vector as input but will ignore everything after the first element.

```
rep(x, each = 2)
```

```
## [1] 1 1 2 2 3 3
```

```
rep(x, each = c(2, 1, 3))
```

```
## Warning in rep(x, each = c(2, 1, 3)): first element used of 'each' argument
```

```
## [1] 1 1 2 2 3 3
```

`length.out` The `length.out` argument in `rep()`, like the `each` argument, will accept a vector of any length, but will ignore all elements beyond the first.

```
rep(x, length.out = 8)
```

```
## [1] 1 2 3 1 2 3 1 2
```

```
rep(x, length.out = c(8, 9))
```

```
## Warning in rep(x, length.out = c(8, 9)): first element used of 'length.out'
## argument
```

```
## [1] 1 2 3 1 2 3 1 2
```

Note: You can use `length.out` to truncate a vector as well.

```
rep(x, length.out = 2)
```

```
## [1] 1 2
```

Class Constructors

We don't often talk about coding efficiency or optimizing your code in this class, preferring to focus instead on teaching you to write clear, correct, and concise code which is easy to write, read, and maintain. One notable exception to this is that it is almost universally better to "pre-allocate" a vector and insert values into it than it is to "grow" a vector (starting from a length-zero vector and repeatedly appending values to the end of it). If you recall my earlier question about how many vectors were created with `x <- c(1, 4, 3)` you can start thinking about why it is very inefficient to "grow" vectors. (*Hint:* you can end up making *a lot* of temporary vectors you'll never see or use.)

So, if we don't want to "grow" a vector how do we pre-allocate a vector of the right type and size? Class constructor functions!

Each class of vector has a constructor function conveniently named the same as the class it constructs! The ones you'll use are (in order of lowest to highest rank in the mode hierarchy (which I'll talk about shortly):

- `logical()`
  - `TRUE`, `FALSE`
  - Fills with: `FALSE`
- `integer()`
  - Any integer valued numeric constant.
  - Fills with: `0L`
  - **NOTE:** `0` is **NOT** an integer, it's integer *valued* but it is stored differently internally. You can specify integer constants by writing a capital "L" at the end of the number (indicating a "long integer" which is 4 bytes) allowing values between -2147483647 and 2147483647. Any mathematical operation which results in an *integer* value beyond this range will return `NA`.
- `numeric()`
  - Any numeric value between $-1.7976931 \times 10^{308}$ and $1.7976931 \times 10^{308}$ (including values with a magnitude as small as $2.2250739 \times 10^{-308}$).
  - Fills with `0`.
  - **Note:** Any operation which generates a numeric value outside this range will return `Inf` or `-Inf` accordingly.

- `character()`
  - Any character string values including Unicode characters (depending on platform/operating system).
  - Fills with: `""` (the **empty string**).
  - **Note:** Be *very* careful making a distinction between the **empty string** and **empty character vector**. In many languages "string" and "character" get bandied about and can often be interchangeable. In R, a **string** refers to the *content* of a length-one character vector. A **character vecctor** *contains* strings. The **empty string** is a length-one character vector which contains zero characters and is represented as a pair of double-quotes (`""`). The **empty character vector** is a character vector which contains zero string elements and is represented by its constructor, `character(0)`.
    Again `""` **is not** `character(0)`, `""` **is** `character(1)`.
    `character(0)` is length-zero, `""` is length-one.

Combining

Once you have some vectors floating around by whatever means, you can combine them into new, more exciting vectors with `c()`.

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

```
## [1] 1 4 3
```

```
(y <- c(2, 2, 7))
```

```
## [1] 2 2 7
```

```
(z <- c(x, y))
```

```
## [1] 1 4 3 2 2 7
```

We will talk more about combining vectors later on when we discuss the **MODE HIERARCHY** wooooooo!

Note: A vector which is the result of combining other vectors will have a length equal to the sum of the lengths of the individual vectors. Think about what this means for combining vectors with length-zero.

Specials

Now, there are a few special values we should discuss which are either vectors themselves or closely related to vectors.

- `NULL`
- `NA` (and `NaN`)
- `Inf` and `-Inf`

`NULL` `NULL` is the ultimate nothing. It doesn't exist as anything other than an indication that there's nothing there. Now, "nothing" is very different in R than "missing" or "unknown" which are represented as `NA` which we'll talk about next.

`NULL` is the only thing you'll encounter in this class which *isn't* a vector (or vector-like object).

`NULL` is a length-zero _non-_vector but if you force it to interact with a vector it will coerce to a length-zero vector of that type. E.g.,

```
NULL & TRUE
```

```
## logical(0)
```

```
NULL + 1L
```

```
## integer(0)
```

```
NULL + 1
```

```
## numeric(0)
```

(Don't worry about the logical operator, we'll get into those next week.)

If you need an object to act as length-zero vector but don't know what class it will eventually become, `NULL` is a safe choice (alternately, you should use `logical(0)` as it has the lowest **MODE HIERARCHY** wooooOOOOoooooOOOoo).

`NA` (and `NaN`) I like to think of `NA` as "I don't know" in R. If you think of logical expressions asking simply "is this statement true?" Most languages only have the option of answering "yes" (`TRUE`) or "no" (`FALSE`), R has the additionally flexibility to say "I don't know" (`NA`).

The default mode/class of `NA` is "logical" (it comes, generally, from trying to answer a logical question), but there are "typed" `NA`s for all of the vector classes.

```
NA_integer_   # integer
```

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

```
NA_real_      # numeric, this is an example of the weirdness of R!
```

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

```
NA_character_ # character
```

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

There's also the weird case of `NaN` which means "Not a Number." This is a *special* kind of unknown.

Note: `NaN` is a type of `NA` but `NA` is not a type of `NaN`, or `NaN` is a *subset* of `NA`.

You'll encounter this when you attempt to do something mathematically invalid. Here are two common examples,

```
0/0
```

```
## [1] NaN
```

```
Inf - Inf
```

```
## [1] NaN
```

Speaking of `Inf`...

`Inf` and `-Inf` `Inf` and `-Inf` are special values representing the idea of infinity. What they really mean though is any number of a larger magnitude than R can handle.

You'll most often bump into these when you are doing some process which lacks "numerical stability." This includes some matrix inversion algorithms and certain optimization techniques.

Coercion (Type-Casting)

R doesn't have a rigid type system which is a dual-edged blade. It allows for a lot of flexibility and creativity in how you write your code, but it can also lead to sloppy habits which create weird, almost impossible problems to debug. There is security in knowing if `x` is an integer vector now it will forever and always remain an integer vector.

Vectors in R coerce to other classes freely upon interaction with each other according to the **MODE HIERARCHY** woooOOOOoooOOooo... okay enough of that, it's time.

Mode Hierarcy

The mode hierachry is the ranking of vector classes according to their relative strength. It is how R decides who wins when two vectors get into a fight. You can also think about it in terms of how "big" each vector class is, that is *how many distinct elements can each class represent*? In order again from weakest/smallest to stongest/largest,

- `NULL` (Special mention, `NULL` is not a vector or a vector class, but it can be coerced to any vector class.) Nothing can be represented by the `NULL` class, it's the smallest possible mode.
- `logical`
  `logical` is the weakest of the proper vector classes, it can only represent three values: `FALSE`, `TRUE`, and `NA` (yes, there are multiple `NA`s, but we'll count them as one here).
- `integer`
  The `integer` mode can represent 4294967295 distinct values in R: 0, 2147483647 positive values, and 2147483647 negative values (system dependent). This might seem like a lot, almost 4.3 billion distinct values, but our next class can handle *several* orders of magnitude more massive numbers as well as countless values between the integer-valued numbers. I *really* don't feel like counting, but if someone wants to calculate exactly how many distinct numeric values there are, I'll doff my cap to you. It's a lot.
- `character`
  The `character` mode can easily represent everything `numeric` can simply by expressing the number as a string. Add in the letters (lower *and* upper cases), countless special characters and unicode characters and the `character` mode quickly seems infinite in scope. It should be little wonder why this is the queen of all the modes.

When a vector is coerced to another mode it does so directly without stopping anywhere along the way (e.g. a logical `FALSE` being coerced to character doesn't make a pit-stop in numeric land to become a `0` first).

In general, you can move *up* the mode hierarchy without losing any information but moving *down* comes with a cost.

Explicit

Each vector class has an associated "casting" function in the form of `as.xxxxxxx()`. You can explicitly convert one vector to another mode using the relevant casting function, e.g.,

```
x <- c(TRUE, FALSE, NA)
as.integer(x)     # up
```

From `logical()`

```
## [1]  1  0 NA
```

```
as.numeric(x)     # up
```

```
## [1]  1  0 NA
```

```
as.character(x)   # up
```

```
## [1] "TRUE"  "FALSE" NA
```

```
x <- c(-1L, 0L, 1L, 2L, 3L)
as.logical(x)     # down
```

From `integer()`

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

```
as.numeric(x)     # up
```

```
## [1] -1  0  1  2  3
```

```
as.character(x)   # up
```

```
## [1] "-1" "0"  "1"  "2"  "3"
```

```
x <- c(-1.2, 0.3, 1.5, 2.0, 3.9)
as.logical(x)     # down
```

From `numeric()`

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

```
as.integer(x)     # down
```

```
## [1] -1  0  1  2  3
```

```
as.character(x)   # up
```

```
## [1] "-1.2" "0.3"  "1.5"  "2"    "3.9"
```

```
x <- c("Mike", "Jake", "Feng", "Bart", "1.4", "TRUE", "F")
as.logical(x)     # down
```

From `character()`

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

```
as.integer(x)     # down
```

```
## Warning: NAs introduced by coercion
```

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

```
as.numeric(x)     # down
```

```
## Warning: NAs introduced by coercion
```

```
## [1]  NA  NA  NA  NA 1.4  NA  NA
```

Notice the loss of information, particularly when R doesn't know how to translate characters into numbers.

Implicit

In addition to being able to explicitly convert vectors from one mode to another, R will "helpfully" do it for you in the background. This is *great* when you want to multiply an vector by a number without needing to worry about if it's an `integer()` or `numeric()`.

```r
a <- 1:3          # an integer vector
typeof(a)
```

```
## [1] "integer"
```

```r
b <- seq(1, 3, 1) # a numeric vector
typeof(b)
```

```
## [1] "double"
```

```r
2 * a
```

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

```r
2 * b
```

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

It's not always great though...

```r
x <- c(12345, 100000, 123456)
x
```

```
## [1]  12345 100000 123456
```

```r
y <- letters[1:3]
y
```

```
## [1] "a" "b" "c"
```

```r
sort(c(x, y))
```

```
## [1] "12345"  "123456" "1e+05"  "a"      "b"      "c"
```

Take a moment and figure out what happened here.

Subsetting

Subsetting a vector in R uses the `[` operator, and is generally done in the form of `x[subset_index_vector]`.

You'll typically think of "subsetting" a vector as doing exactly what it sounds like, creating a smaller subset vector from the parent vector. e.g.,

```r
x <- 11:16
x[c(1, 3, 2)]
```

```
## [1] 11 13 12
```

And, that's fine, but there are lots of other ways subsetting can be used.

You can select multiple copies of some of the elements,

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

```
## [1] 12 12 14 12
```

You can even make a "subset" vector longer than the original!

```
x[rep(seq_along(x), x)]
```

```
##  [1] 11 11 11 11 11 11 11 11 11 11 11 12 12 12 12 12 12 12 12 12 12 12 12 13 13
## [26] 13 13 13 13 13 13 13 13 13 13 14 14 14 14 14 14 14 14 14 14 14 14 14 14
## [51] 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 16 16 16 16 16 16 16 16 16 16
## [76] 16 16 16 16 16 16
```

You might do this more practically when we get to plotting and you want to assign different colors to different observations.

```
colors <- c("red", "green", "blue")
classes <- c(3L, 2L, 3L, 2L, 2L, 2L, 1L, 2L, 1L, 3L, 3L, 3L, 1L, 1L, 3L)
colors[classes]
```

```
##  [1] "blue"  "green" "blue"  "green" "green" "green" "red"   "green" "red"
## [10] "blue"  "blue"  "blue"  "red"   "red"   "blue"
```

Positive

What we've seen so far has been "positive" subsetting, we have a vector `x`, and we identify the elements in `x` we want to keep and in the order we want them. This is by far the most common way to subset and is the only way to keep multiples of the same element or create a subset larger than the original.

```
x
```

```
## [1] 11 12 13 14 15 16
```

```
x[c(1, 3, 4)]
```

```
## [1] 11 13 14
```

But we can also tell R what we don't want...

Negative

If the index values you subset with are *negative* R will return the vector with those elements dropped.

```
x
```

```
## [1] 11 12 13 14 15 16
```

```
x[-1]
```

```
## [1] 12 13 14 15 16
```

```
x[-2:-4]
```

```
## [1] 11 15 16
```

```
x[-2:4] # why doesn't this work?
```

```
## Error in x[-2:4]: only 0's may be mixed with negative subscripts
```

Look at what happened with the last attempt. This is a common error students encounter, you can't mix positive and negative subscripts, take a few moments to reason through why that is.

Which brings us to what I call "oddities."

Oddities

There are four types of unusual indices you might run into (often accidentally and you'll need to debug or make your code more defensive, but you might also use these deliberately as a cute way to accomplish something which might otherwise be challenging).

0 R is a one-indexed language, so the first element is element `1`. if you have previous coding experience this will probably frustrate and anger you. It's okay, you'll survive! If you're completely new to programming, you'll probably prefer this and find it intuitive. Regardless, as R is primarily focus on analyzing data, this convention makes much more sense.

For reasons lost to time, R returns a length-zero vector with a zero index.

```
x[0]
```

```
## integer(0)
```

This vector will always be of the same type as `x` so it is a very nice way to create an empty vector the same type as another when you won't know ahead of time what the required type will be,

```
y <- x[0]
y
```

```
## integer(0)
```

Length-zero A length-zero subset index vector behaves exactly like the `0` index.

```
x[NULL]
```

```
## integer(0)
```

NA An `NA` index returns `NA`

```
x[c(1, NA, 3)]
```

```
## [1] 11 NA 13
```

Out-of-Bounds Out of bounds indices (indicies greater than the length of the vector) also return `NA`.

```
x[8]
```

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

This behavior is one of the reasons I **VERY STRONGLY** recommend using the **extraction** operator (`[[`) rather than the **subset** operator (`[`) when you are working with individual values in a vector. I'll talk about this briefly later.

Recycling

From the *R Language Definition section 3.3.1, Recycling rules*

> If one tries to add two structures with a different number of elements, then the shortest is recycled to length of longest. That is, if for instance you add c(1, 2, 3) to a six-element vector then you will really add c(1, 2, 3, 1, 2, 3). If the length of the longer vector is not a multiple of the shorter one, a warning is given.

Implicit

Implicit recycling will happen anytime you perform an operation jointly on two different length vectors.

```
x <- 1:3
y <- 4:9
x + y
```

```
## [1]  5  7  9  8 10 12
```

```
x * y
```

```
## [1]  4 10 18  7 16 27
```

```
x^y
```

```
## [1]     1    32   729     1   256 19683
```

Think about the following expression,

```
x + 1
```

```
## [1] 2 3 4
```

What's happening here?

Clearly we are adding `1` to each element of `x`, but is it the *same* `1`?

It's an incredibly nit-picky distinction, but I believe thinking about it will aid in cementing this concept in your minds.

When we do `x + 1` which is `c(1, 2, 3) + 1` we're not taking that *same* `1` and adding it to each element of `x`. Internally R *recycles* the `1` to be the same length as `x`, so what you're really doing is `c(1, 2, 3) + c(1, 1, 1)`!

Does it matter? No, not at all! But, I hope pointing this out helps you internalize the recycling concept.

Perfect vs Imperfect Now, I want to talk about the idea of *perfect* vs *imperfect* recycling. When the length of the longer vector is a multiple of the length of the shorter vector, I will refer to this as **perfect recycling**. Perfect recycling doesn't throw any **warning()** messages which is nice from an end user perspective (warnings tend to scare users and novice programmers) but, this can be dangerous because you won't know if recycling is happening or not. e.g. let's say I have a function `increment_by()` which I use to increase the value of every element of a vector `x` by an increment `by` (with a default value of `1`),

```
increment_by <- function(x, by = 1) {
  x + by
}
x <- 1:4
increment_by(x)
```

```
## [1] 2 3 4 5
```

```
increment_by(x, 2)
```

```
## [1] 3 4 5 6
```

Great!

Now, let's say a user is using this as part of a function they are writing and they take the value for `by` from a computed variable `y` as the range of `x` not realizing `range()` computes a length-two vector containing the minimum and maximum of `x`, e.g.,

```
# What the user meant
# y <- diff(range(x))

# What the user wrote
y <- range(x)
increment_by(x, y)
```

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

If `length(x)` is a multiple of `length(y)` the user will not receive any indication something is amiss.

NOTE: This is an artificial example, but this would be resolved by adding some input validation to your `increment_by()` function. Input validation will be discussed in Chapter 4.

What I'd like you to come away from this with is that, while you need not be paranoid about it, you should be keenly aware that R is really green and is *always* looking for things it can recycle.

Explicit

Now, you don't have to rely on R recycling things for you. You can and *should* take the initiative and do your own recycling.

You can use the `rep()` function for this. Let's say we have a function `add_xy()` which, simply takes an `x` and `y` value and adds them together (you might notice this is functionally identical to our `increment_by()` function).

```
add_xy <- function(x, y) {
  x + y
}
```

Further, let us say we (for whatever reason) don't want to see a warning when R recycles imperfectly.

```
add_xy(1:3, 1:4)
```

```
## Warning in x + y: longer object length is not a multiple of shorter object
## length
```

```
## [1] 2 4 6 5
```

We can do this by handling the recycling ourselves with `rep()`! First we will need to know how long our output length is going to be, which will be the maximum of the two lengths. Then we just need to extend each vector to be that length. Repeating each vector a number of times won't work since the `times` argument doesn't work with fractional values but, thankfully, in addition to the optional arguments `each` and `times`, `rep()` will also accept a `length.out` argument like `seq()` does.

```
add_xy <- function(x, y) {
  output_length <- max(length(x), length(y))
  x <- rep(x, length.out = output_length)
  y <- rep(y, length.out = output_length)
  x + y
}
add_xy(1:3, 1:4)
```

```
## [1] 2 4 6 5
```

And the warning message is gone!

Best Practices

In general I recommend handling recycling yourself. When we get to Chapter 4 I will talk at length about how functions you write should (almost) never throw warnings or errors generated by {base} R functions. {base} R errors and warnings aren't always that helpful and can often be cryptic or indecipherable to the end users of your functions. For now you should understand it is better to usually not rely on R implicitly recycling on your behalf.

Extracting (Bonus Topic)

There is a second operator related to the subsetting operator (`[`) called the **extraction** operator (`[[`). We will discuss this much more in Chapter 6, and I only mention it now to make you aware.

The main differences between `[` and `[[` are,

- With `x[n]`, `n` may be a vector of values of any length while with `x[[n]]`, `n` **must** be a length-one vector or it will throw an error.

```
x <- 1:4
x[2:3]
```

```
## [1] 2 3
```

```
x[[2:3]]
```

```
## Error in x[[2:3]]: attempt to select more than one element in vectorIndex
```

- `x[n]` creates an object which is a smaller version of whatever `x` is and `x[[n]]` creates an object which is whatever the third element of `x` is. As the only objects you have now are vectors and (almost) everything in R is a vector this might seem to be a distinction without a difference, but I promise this will make a lot more sense later.
- `x[n]` will return `NA` if `n > length(x)` while `x[[n]]` will throw an error if you attempt to extract an out-of-bounds index.

```
x <- 1:4
x[5]
```

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

```
x[[5]]
```

```
## Error in x[[5]]: subscript out of bounds
```

- `x[n]` will return a length-zero vector of the same type as `x` if you `n` is `0` or length-zero, while `x[[n]]` will throw an error.

```
x <- 1:4
x[0]
```

```
## integer(0)
```

```
x[integer(0)]
```

```
## integer(0)
```

```
x[[0]]
```

```
## Error in x[[0]]: attempt to select less than one element in get1index <real>
```

```
x[[integer(0)]]
```

```
## Error in x[[integer(0)]]: attempt to select less than one element in get1index
```

Best Practices

In general you *want* your functions to throw errors. I know, I know... This sounds like CrAzY tAlK, but hear me out. You want to know *immediately* when the reality of what is happening inside your code diverges from your expectations. Errors (and warnings) are how you (and your end users) get this information. We will talk about this much more in Chapter 4, but for now you should understand *errors are a good thing*!

Writing your code in a strict way so it doesn't fail invisibly is a best practice. So, if you are writing code where you need to get exactly one element out of a vector `[[` is preferred to `[`. I'll give you plenty of examples for this when we get to Chapter 4.

Vectorization

Vectorization is simply the practice (or thought process) of doing something to many things simultaneously.

R is *natively* vectorized, something it inherited from its FORTRAN roots.

Native Vectorization

We've already seen it in practice several times, but here it is again,

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

```
## [1] 3 3 7
```

R has no problem doing this **vectorized** addition. It's a core principle of the language: > (almost) everything is a vector and (almost) every function is vectorized

In *most* other languages, if you wanted to do this same operation, you'd need to loop through the indices of `x` and `y` one at a time, extract the relevant elements of each, add those together, store the result in the corresponding index of a third object, then finally return the output.

R spoils us! As R programmers we are free to think in terms of vectors.

What do we want to do? Element-wise addition of `x` and `y`? Easy! Just add them together `x + y`. Done!

If R is your first language, you are one of the lucky ones, you get to learn vectorization from the start! If you know one (or more) other languages try to avoid being rigid in your thinking and insisting on using familiar paradigms (loops).

> Of course, someone has to write loops. It doesn't have to be you. — Jenny Bryan

We'll talk about loops vs vectorization more in, wait for it... Chapter 4, but vectorized functions *are* loops – just at a lower level than R. They're often (but by no means always) faster than high level loops and vectorized code is usually cleaner and easier to write, read, understand, and debug *especially* once you develop your *vectorized mind*.

So, the **best practice** in R is to favor vectorization.

(Almost) all {base} R functions are vectorized, that is they can take vectors (or vector-like objects) as inputs and return vectors (or vector-like objects) as outputs.

When we encounter situations where the above *is not* the case, we can restore order to the world with `vapply()`!

`vapply()`

`vapply()` allows us to **apply** a function to every element of a vector (or vector-like object).

The canonical example given in the notes is the `isTRUE()` function which asks, "is this thing I am giving you exactly the same as `TRUE`?"

So, `isTRUE(TRUE)` will return `TRUE`, but `isTRUE(c(TRUE, TRUE))` will return `FALSE`. If we want to know if the individual elements are `TRUE`, that is if we want to ask, "for each of the elements in this thing I am giving you, are they exactly the same as `TRUE`?" We can create a vectorized version using `vapply()`.

`vapply()` takes the following,

- A vector (or vector-like object) on which the function will be "vectorized over."
- A function to apply to each element of the vector (or vector-like object), this can be either
  - a **named function** (either a built-in or written by you)
  - an **anonymous function** defined inside the call to `vapply()`
- A **prototype** value showing `vapply()` *exactly* what the output of each function call will be.
- Any additional arguments which will be passed to the applied function.

Anonymous Functions An **anonymous function** is a function which hasn't been stored in another R object. It *is* an R object, but an ephemeral one. You'll usually use **anonymous functions** when the function you are writing isn't important enough to keep around. Here's an artificial example:

Let's say we want a function `var_n()` which takes a number `n` and computes the variance of a length-`n` regular sequence.

```
var_n <- function(n) {
  var(seq_len(n))
}
var_n(3)
```

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

```
var_n(7)
```

```
## [1] 4.666667
```

Great, but... it's not vectorized! We cannot get results for multiple values of `n` at once,

```
var_n(3:10)
```

```
## Warning in seq_len(n): first element used of 'length.out' argument
```

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

What to do!?

Well, we can of course use `vapply()` with our existing `var_n()` function,

```
vapply(3:10,
       var_n,
       numeric(1))
```

```
## [1] 1.000000 1.666667 2.500000 3.500000 4.666667 6.000000 7.500000 9.166667
```

But, if we didn't have the `var_n()` function yet, we can just take the right-hand side of the assignment,

```
function(n) {
  var(seq_len(n))
}
```

```
## function(n) {
##   var(seq_len(n))
## }
```

and use that as our `FUN` argument,

```
vapply(3:10,
       function(n) {
         var(seq_len(n))
       },
       numeric(1))
```

```
## [1] 1.000000 1.666667 2.500000 3.500000 4.666667 6.000000 7.500000 9.166667
```

this is how we use an **anonymous function**, we're using an object which isn't named.

Now, we can use this idea to write a **vectorized** version of `var_n()` by changing the body of `var_n()` to be this call to `vapply()`,

```
var_n <- function(n) {
  vapply(n,                 # NOTE, we change this to be n
       function(m) {        # Leaving this as n would be fine
         var(seq_len(m))    # but it is more readable if we
       },                   # choose an alternate.
       numeric(1))
}
```

Now our function can take its rightful place in this vectorized world,

```
var_n(3:10)
```

```
## [1] 1.000000 1.666667 2.500000 3.500000 4.666667 6.000000 7.500000 9.166667
```

Prototypes The **prototype** value you pass to the `FUN.VALUE` argument must be exactly of the same form as the return value from your applied function, so you have to be very, very careful sometimes.

Generally, the best practice is to use one of the vector class constructors and a value for the expected length (e.g. `logical(1)`, `numeric(8)`, etc). But, this won't always be possible.

Consider the following, very contrived, example of the function `self()`, which just returns whatever you give it,

```
self <- function(x) {
  x
}
```

If we were going to use this function in `vapply()`, what should out prototype be? Let's choose `integer(1)` and see how that works,

```
x <- 1:4
vapply(x,
       self,
       integer(1))
```

```
## [1] 1 2 3 4
```

Perfect, no problems! But...

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

```
## Error in vapply(x, self, integer(1)): values must be type 'integer',
##  but FUN(X[[1]]) result is type 'double'
```

Uh-oh...

What to do?

Well, we can just use any single element of `x` as the prototype,

```
x <- c(1, 2, 3, 4)
vapply(x,
       self,
       x[1])
```

```
## [1] 1 2 3 4
```

So, if we wanted to make a vectorized version of `self()` (note that this doesn't make a lot of sense right now since `self()`is already vectorized and vectorization doesn't really mean anything in this context, just go with me here, I already said this example is very contrived), we might do something like,

```
self <- function(x) {
  vapply(x,
         function(y) {
           y
         },
         x[1])
}
self(1:4)
```

```
## [1] 1 2 3 4
```

```
self(c(1, 2, 3, 4))
```

```
## [1] 1 2 3 4
```

```
self(integer(0))
```

```
## integer(0)
```

Additional Arguments (... Intro) When you apply a function to the elements of a vector (or vector-like object) with `vapply()`, those elements are passed to (generally) the first argument of the function. It's possible to send other arguments to the function by putting them in the `...` argument of `vapply()`. See the following,

```
f <- function(x, y, z) {
  x * y^z
}
```

When we use `vapply()` any arguments we supply *after* the **prototype** are collected by the `...` argument and passed along to the applied function. So, if we write,

```
vapply(1:4,
       f,
       numeric(1),
       2,
       3)
```

```
## [1]  8 16 24 32
```

The `2` and the `3` are passed along to `y` and `z` respectively. **NOTE:** R is matching by *position* here as everything is un-named. What if we wanted to do this, but vectorize over `y` instead of `x`? We *could* change the order of the arguments in our function like so,

```
f <- function(y, x, z) {
  x * y^z
}
```

so that the first arg is `y`, but that's ugly and bad and makes the function weird and unintuitive to use more generally. Better is to simply name the additional arguments we use inside the `...`,

```
vapply(1:4,
       f,
       numeric(1),
       x = 2,
       y = 3)
```

```
## [1]   6  18  54 162
```

Results If the **prototype** is of length-one, the output of a `vapply()` function call will be a vector the same length as the input vector.

If the **prototype** is a vector with length greater than one, the result will be a two-dimensional array called a **matrix**. We'll discuss matrices in detail in chapter 5, but for now know the result of each call to the applied function will be stored in the corresponding column of the output matrix. For now, see this example,

```
vapply(1:10,
       function(n) {
         summary(seq_len(n))
       },
       numeric(6))
```

```
##         [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## Min.       1 1.00  1.0 1.00    1 1.00  1.0 1.00    1  1.00
## 1st Qu.    1 1.25  1.5 1.75    2 2.25  2.5 2.75    3  3.25
## Median     1 1.50  2.0 2.50    3 3.50  4.0 4.50    5  5.50
## Mean       1 1.50  2.0 2.50    3 3.50  4.0 4.50    5  5.50
## 3rd Qu.    1 1.75  2.5 3.25    4 4.75  5.5 6.25    7  7.75
## Max.       1 2.00  3.0 4.00    5 6.00  7.0 8.00    9 10.00
```

When Can We *NOT* use `vapply()`? `vapply()` won't work when the outputs of each call to the applied function aren't the same. See an example below,

```
vapply(2:5,
       seq,
       integer(2))
```

```
## Error in vapply(2:5, seq, integer(2)): values must be length 2,
##  but FUN(X[[2]]) result is length 3
```

Here, an error is thrown on the first input where the output doesn't match the prototype. There, of course, isn't a prototype which will match all of the outputs making `vapply()` inappropriate for this task. In Chapter 6 we will cover other members of the **apply family of functions** which are more flexible in this regard.

Practice

`my_var()`

Write a version of a variance function which takes a data vector `x` and a logical value `sample` indicating whether we should compute the sample variance or the population variance of `x`.

Criteria

1. Be mathematically correct.
2. Allow option for sample or population variance.

Pseudocode We will just use the mathematical expressions for variance as our pseudocode here:

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2 \tag{1}$$

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2 \tag{2}$$

```
IF sample
    compute sample variance s^2
ELSE
    compute population variance s^2
```

This is tricky and *maaaaayyyybe* a bit unfair, you don't have access to `if/else` statements yet (Chapter 4), so how can you possibly be expected to implement something which requires a conditional? Well... we're going to get creative that's how!

Let's look again at the expressions for variance,

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2 \tag{3}$$

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2 \tag{4}$$

They're *almost* identical... the only difference is the denominators.

We can tweak them a bit though to make the form of them exactly the same by fixing the denominator of the sample variance to include the missing subtraction, making use of the additive identity `0`.

$$\sigma^2 = \frac{1}{n-0}\sum_{i=1}^{n}(x_i - \bar{x})^2 \tag{5}$$

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2 \tag{6}$$

So, now we can combine these into one equation.

$$\text{variance} = \frac{1}{n - (0 \text{ if population or } 1 \text{ if sample})}\sum_{i=1}^{n}(x_i - \bar{x})^2 \tag{7}$$

What can you think of which has a value of either `0` or `1`?

Hint: what does the function we are writing have which we haven't explicitly used yet?

The `sample` argument!

Since `sample` is a logical value and the logical values `FALSE` and `TRUE` when cast to integers become `0` or `1` respectively we can use our `sample` argument as the thing to subtract in the denominator.

$$\text{variance} = \frac{1}{n - \texttt{sample}} \sum_{i=1}^{n} (x_i - \bar{x})^2 \tag{8}$$

Sweet, let's write this thing!

```
my_var <- function(x, sample = TRUE) {
  sum((x - mean(x))^2) / (length(x) - sample)
}
x <- 1:10
var(x)
```

Code

```
## [1] 9.166667
```

```
my_var(x)
```

```
## [1] 9.166667
```

```
my_var(x, sample = FALSE)
```

```
## [1] 8.25
```

`seq_sum()`

Write the function `seq_sum()` which computes the sum of the regular sequence of the first `n` natural numbers (integers greater than 0). Then, extend the function to accept vectors for `n` which are longer than one.

Pseudocode We should start with the basic problem of doing one sequence sum first,

```
make a length n sequence
add all the elements of that sequence
RETURN the sum value
```

Then we can extend it to be vectorized,

```
FOR each element n_i of n
    make a length n_i sequence
    add all the elements of that sequence
RETURN all of the sequence sums
```

Code We'll again start with the basic case where `n` is a length-one vector. A direct, statement by statement implementation of the pseudocode might be something like,

```
seq_sum <- function(n) {
  v <- seq_len(n)
  sum(v)
}
```

However, it's (again *generally*) a best practice not to create variables you'll only use once (as we do with `v`) *unless* doing so makes your code more readable. **NOTE:** When one thing is more readable than another is often a personal taste. In this class you should use this as your guide: 1. Follow the tidyverse style guide 2. Don't create single use variables 3. Comments should only explain *why* never *what*

So, better would be,

```
seq_sum <- function(n) {
  sum(seq_len(n))
}
seq_sum(4)
```

```
## [1] 10
```

Now to **vectorize** it,

```
seq_sum <- function(n) {
  vapply(n,
         function(m) {
           sum(seq(m))
         },
         numeric(1))
}
seq_sum(1:10)
```

```
##  [1]  1  3  6 10 15 21 28 36 45 55
```

And, that's it... go have fun!