Coding Lab: Vectors and data types

Ari Anisfeld

Summer 2020

Vectors[1]

Vectors are the foundational data structure in R.

Here we will discuss how to:

- construct vectors and tibbles
- do vectorized math and computations
- deal with missing values
- work with vectors of different data types

[1]Technically, I'm talking about "atomic vectors".

Vectors

Vectors store an arbitrary[2] number of items of the same type.

```r
# numeric vector of length 6
my_numbers <- c(1, 2, 3, 4, 5, 6)

# character vector of length 3
my_characters <- c("public", "policy", "101")
```

[2]Within limits determined by hardware

Vectors

In R, nearly every object you will work with is a vector

```r
# vectors of length 1
tis_a_vector <- 1919
technically_a_logical_vector <- TRUE
```

The `c()` function combines vectors

```r
c(c(1, 2, 3), c(4, 5, 6))
```

```
## [1] 1 2 3 4 5 6
```

```r
c(tis_a_vector, 1920)
```

```
## [1] 1919 1920
```

Creating vectors

There are some nice shortcuts for creating vectors.

```
c("a", "a", "a", "a")
```

```
## [1] "a" "a" "a" "a"
```

```
rep("a", 4)
```

```
## [1] "a" "a" "a" "a"
```

Try out the following:

```
rep(c("a", 5), 10)
rep(c("a", 5), each = 10)
```

Creating vectors

There are also several ways to create vectors of sequential numbers:

```r
c(2, 3, 4, 5)
```

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

```r
2:5
```

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

```r
seq(2, 5)
```

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

Creating random vectors

Create random data following a certain distribution

```
(my_random_normals <- rnorm(5))
```

```
## [1] -0.09370775 -0.55002432 -1.32800330  0.66192728 -1.
```

```
(my_random_uniforms <- runif(5))
```

```
## [1] 0.1834452 0.2809709 0.7093054 0.7176520 0.8755603
```

Creating empty vectors of a given type

Create empty vectors of a given type[3]

```r
# 1 million 0
my_integers <- integer(1e6)

# 40K ""
my_chrs <- character(4e5)
my_chrs[1:10]
```

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

[3]We'll discuss what types are soon.

Binary operators are vectorized

We can do math with vectors!

```r
my_numbers <- 1:6

# this adds the vectors item by item
my_numbers + my_numbers
```

```
## [1]  2  4  6  8 10 12
```

```r
# this adds 6 to each object  (called recycling)
my_numbers + 6
```

```
## [1]  7  8  9 10 11 12
```

Vectorized functions built into R

Some vectorized functions operate on each value in the vector and return a vector of the same length[4]

- These are used with `mutate()`

```r
a_vector <- rnorm(100)
sqrt(a_vector) # take the square root of each number
log(a_vector) # take the natural log of each number
exp(a_vector) # e to the power of each number
round(a_vector, 2) # round each number

str_to_upper(a_chr_vector) # make each chr uppercase
str_replace(a_chr_vector, "e", "3")
```

[4] try it out yourself! use `?func` to learn more

Warning: Vector recycling

Be careful when operating with vectors. What's happening here?

```
a <- 1:6 + 1:5
```

```
## Warning in 1:6 + 1:5: longer object length is not a mult
## length
```

```
a
```

```
## [1]  2  4  6  8 10  7
```

Warning: Vector recycling

Be careful when operating with vectors. If they're different lengths, the shorter vector starts from it's beginnig (`6 + 1 = 7`).

```
a <- c(1, 2, 3, 4, 5, 6) + c(1, 2, 3, 4, 5)
```

```
## Warning in c(1, 2, 3, 4, 5, 6) + c(1, 2, 3, 4, 5): longe
## multiple of shorter object length
```

```
# 1 + 1,
# 2 + 2,
# 3 + 3,
# 4 + 4,
# 5 + 5,
# !!!6 + 1!!! Recycled.
a
```

```
## [1]  2  4  6  8 10  7
```

Binary operators are vectorized

We can do boolean logic with vectors!

```r
my_numbers <- 1:6
# behind the scenes 4 is recycled
# to make c(4, 4, 4, 4, 4, 4)
my_numbers > 4
```

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

```r
my_numbers == 3
```

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

Binary operators are vectorized

We can do boolean logic with vectors!

```
my_numbers <- 1:6

# c(1, 2, 3, 4, 5, 6) >  c(1, 1, 3, 3, pi, pi)
# occurs item by item
my_numbers > c(1, 1, 3, 3, pi, pi)
```

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

Functions that reduce vectors

Others take a vector and return a summary[5]

- These are used with `summarize()`

```
sum(a_vector)  # add all the numbers
median(a_vector) # find the median
length(a_vector) # how long is the vector
any(a_vector > 1) # TRUE if any number in a_vector > 1

a_chr_vector <- c("a", "w", "e", "s", "o", "m", "e")
paste0(a_chr_vector) # combine strings
```

[5] try it out yourself! use `?func` to learn more

Tibble columns are vectors

We can create tibbles manually

- To test out code on a simpler tibble
- To organize data from a simulation

```r
care_data <- tibble(
  id = 1:5,
  n_kids = c(2, 4, 1, 1, NA),
  child_care_costs = c(1000, 3000, 300, 300, 500),
  random_noise = rnorm(5, sd = 5)*30
)
```

Subsetting

Three ways to pull out a column as a vector.[6]

```
# tidy way
care_data %>% pull(n_kids)
```

```
## [1]  2  4  1  1 NA
```

```
# base R way
care_data$n_kids
```

```
## [1]  2  4  1  1 NA
```

```
# base R way
care_data[["n_kids"]]
```

```
## [1]  2  4  1  1 NA
```

[6]See Appendix for more on subsetting

Subsetting

Two ways to pull out a column as a tibble

```
# tidy way
care_data %>% select(n_kids)
```

```
## # A tibble: 5 x 1
##   n_kids
##    <dbl>
## 1      2
## 2      4
## 3      1
## 4      1
## 5     NA
```

```
# base R way
care_data["n_kids"]
```

```
## # A tibble: 5 x 1
##   n_kids
##    <dbl>
```

Type issues

Sometimes you load a data set, write code that makes sense and get an error like this:

```
care_data %>%
  mutate(spending_per_child = n_kids / child_care_costs)
```

```
Error in n_kids/child_care_costs : non-numeric
argument to binary operator
```

Type issues

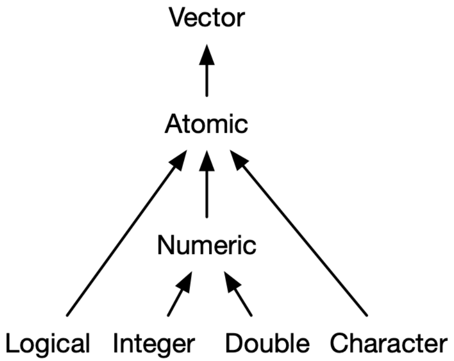
```r
glimpse(care_data)
```

```
## Observations: 5
## Variables: 4
## $ id               <int> 1, 2, 3, 4, 5
## $ n_kids           <dbl> 2, 4, 1, 1, NA
## $ child_care_costs <dbl> 1000, 3000, 300, 300, 500
## $ random_noise     <dbl> 108.09427, 167.97536, -22.5066
```

Data types

R has four primary types of atomic vectors

- these determine how R stores the data (technical)



[7]

[7] Image from https://adv-r.hadley.nz/vectors-chap.html

Data types

Focusing on the types, we have:

```r
# logical, also known as booleans
type_logical <- FALSE
type_logical <- TRUE

# integer and double, together are called: numeric
type_integer <- 1L
type_double <- 1.0

type_character <- "abbreviated as chr"
type_character <- "also known as a string"
```

Testing types

```r
a <- "1"
typeof(a)
```

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

```r
is.integer(a)
```

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

```r
is.character(a)
```

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

Testing types

In our example:

```r
typeof(care_data$child_care_costs)
```

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

```r
typeof(care_data$n_kids)
```

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

Type coercion

The error we got when we tried `a + b` was because `a` is a character. We can reassign types on the fly:

```r
a <- "4"
as.integer(a) + 3
```

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

```r
as.numeric(a) + 3
```

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

NAs introduced by coercion

The code produces a warning! Why? R does not know how to turn the string "unknown" into an integer. So, it uses `NA` which is how R represents missing or unknown values.

```r
as.integer("Unknown")
```

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

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

NAs are contagious

```r
NA + 4
```

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

```r
max(c(NA, 4, 1000))
```

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

Type coercion

To address our problem, we use `mutate()` and `as.integer()` to change the type of `n_kids`

```
care_data %>%
  mutate( n_kids = as.integer(n_kids),
          spending_per_kid = child_care_costs / n_kids)
```

```
## # A tibble: 5 x 5
##      id n_kids child_care_costs random_noise spending_pe
##   <int>  <int>            <dbl>        <dbl>
## 1     1      2             1000        108.
## 2     2      4             3000        168.
## 3     3      1              300        -22.5
## 4     4      1              300        -65.4
## 5     5     NA              500        110.
```

Automatic coercion (Extension material to be discussed live)

Some type coercion is done by R automatically:

```r
# paste0() is a function that combines two chr into one
paste0("str", "ing")
```

```
## [1] "string"
```

```r
paste0(1L, "ing")
```

```
## [1] "1ing"
```

`1L` is an `int`, but R will coerce it into a `chr` in this context.

Automatic coercion

Logicals are coercible to numeric or character. This is very useful!

What do you think the following code will return?

```r
TRUE + 4
FALSE + 4
paste0(FALSE, "?")
mean(c(TRUE, TRUE, FALSE, FALSE, TRUE))
```

Automatic coercion

```r
TRUE + 4
```

```
## [1] 5
```

```r
FALSE + 4
```

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

```r
paste0(FALSE, "?")
```

```
## [1] "FALSE?"
```

```r
mean(c(TRUE, TRUE, FALSE, FALSE, TRUE))
```

```
## [1] 0.6
```

NAs are contagious, redux.

```r
b <- c(NA, 3, 4, 5)
sum(b)
```

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

NAs are contagious, redux.

Often, we can tell R to ignore the missing values.

```r
b <- c(NA, 3, 4, 5)
sum(b, na.rm = TRUE)
```

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

Subsetting vectors

Use [[for subsetting a single value

```
# letters is built into R and has lower case letters from
# get the third letter in the alphabet
letters[[3]]
```

```
## [1] "c"
```

Use [for subsetting multiple values

```
# get the 25th, 5th and 19th letters in the alphabet
letters[c(25,5,19)]
```

```
## [1] "y" "e" "s"
```

Subsetting vectors

Using a negative sign, allows subsetting everything except th

```
my_numbers <- c(2, 4, 6, 8, 10)
# get all numbers besides the 1st
my_numbers[-1]
```

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

```
# get all numbers besides the 1st and second
my_numbers[-c(1,2)]
```

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

We can also subset with booleans

```
# get all numbers where true
my_numbers[c(TRUE, FALSE, FALSE, TRUE, FALSE)]
```

```
## [1] 2 8
```

```
my_numbers[my_numbers > 4]
```

Subsetting recommendations

I recommend sticking with the tidy version when working with tibbles and data.

- Tidyverse functions will cover nearly all of your data processing needs.
- The `[` and `[[` subsetting have a lot of subtle and unexpected behavior.
- If you find yourself doing "programming"" in R then it is worth revisiting subsetting in `adv-r`

Example: Using vectors to calculate a sum of fractions

Use R to calculate the sum

$$\sum_{n=0}^{10} \frac{1}{2^n}$$

How would you translate this into code?

Example: Using vectors to calculate a sum of fractions

We go from math notation

$$\sum_{n=0}^{10} \frac{1}{2^n}$$

to R code:

```r
numerators <- rep(1, 11)
denominators <- 2 ^ c(0:10)

sum(numerators/denominators)
```

```
## [1] 1.999023
```

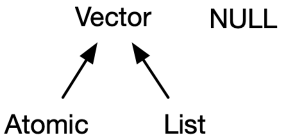
Recap: Vectors and data types

We discussed how to:

- Create vectors and tibbles for various circumstances
- Do vectorize operations and math with vectors (we implicitly did this with `mutate`)
- Subset tibbles (we explicitly did this with `select` and `filter`)
- Understand data types and use type coercion when necessary.

Technical note: Atomic vectors vs lists

- Atomic vectors have a single type.
- Lists can hold data of multiple types.[8]



[8]This is beyond our scope, but lists can be thought of as a vector of pointers. The interested student can read more at https://adv-r.hadley.nz/

Technical note: a Lists holding multiple types.

```r
a_list <- list(1L, "fun", c(1,2,3))
typeof(a_list)
```

```
## [1] "list"
```

```r
typeof(a_list[[1]])
```

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

```r
typeof(a_list[[2]])
```

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

```r
typeof(a_list[[3]])
```

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