Lecture 2: Vectors, Lists and Data Frames (base R)

Harris Coding Camp – Accelerated Track

Summer 2022

Todays class

Build foundational skills with

- Vectors
- Data types
- data.frame and tibble

Vectors

Vectors

Vectors are the foundational data structure in R.

Here we will discuss how to:

- construct vectors, lists and data frames
- do vectorized math and computations
- deal with missing values (`NA`)
- work with vectors of different data types

Vectors

Vectors store an arbitrary[1] number of items of the *same* type.

`c()` is used to create a vector with explicitly given items.

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

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

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

```
## [1] "public" "policy" "101"
```

[1] Within limits determined by hardware

Vectors

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

```
# vectors of length 1
i_am_a_vector <- 1919
as_am_i <- TRUE

is.vector(i_am_a_vector)
```

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

```
is.vector(as_am_i)
```

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

```
# Some objects are not vectors e.g. functions
is.vector(mean)
```

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

Vectors

Thus the `c()` function combines vectors

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

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

```
y <- c(x, 2022)
y
```

```
## [1]    1    2    3    4    5    6 2022
```

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
```

```r
seq(from = 2, to = 5, by = 1)
```

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

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), 4)
rep(c("a", 5), each = 4)
```

Vectors

```r
rep(c("a", 5), 4)
```

```
## [1] "a" "5" "a" "5" "a" "5" "a" "5"
```

```r
rep(c("a", 5), each = 4)
```

```
## [1] "a" "a" "a" "a" "5" "5" "5" "5"
```

Creating empty vectors of a given type

Some commonly used different vector modes/types:[2]

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

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

```
# Vector of mode 'character' with 5 elements
vector(mode="character", length = 5)
```

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

```
# Same thing, but using the constructor directly
character(5)
```

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

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

Creating empty vectors of a given type

```r
# 1 million 0s
my_integers <- integer(1e6)
# 40K FALSEs
my_lgl <- logical(4e5)
my_lgl[1:10]
```

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

Adding elements to an existing vector

```r
z <- c("Bo", "Cynthia", "David")
z
```

```
## [1] "Bo"       "Cynthia" "David"
```

```r
z <- c(z, "Ernesto")
z
```

```
## [1] "Bo"       "Cynthia" "David"   "Ernesto"
```

```r
z <- c("Amelia", z)
z
```

```
## [1] "Amelia"  "Bo"       "Cynthia" "David"   "Ernesto"
```

Examining Vectors

```
length(z)
```

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

```
summary(z)
```

```
##     Length     Class      Mode
##          5 character character
```

Accessing Elements by Index

```
z[3]
```

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

```
z[2:4]
```

```
## [1] "Bo"      "Cynthia" "David"
```

We can reassign accessed values too.

```
z[1] <- "Arthur"
z[c(1,3)]
```

```
## [1] "Arthur"  "Cynthia"
```

Accessing Element by Logical Vector

```
c(TRUE, TRUE, FALSE, FALSE, TRUE)
```

```
z[c(TRUE, TRUE, FALSE, FALSE, TRUE)]
```

```
## [1] "Arthur"  "Bo"      "Ernesto"
```

Removing items from a vector

Using a negative sign, allows subsetting everything except the selected one(s):

```
my_letters <- c("a", "b", "c", "d", "e")
# get all numbers besides the 1st
my_letters[-1]
```

```
## [1] "b" "c" "d" "e"
```

```
# get all numbers besides the 2nd and 4th
my_letters[-c(2,4)]
```

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

Creating random vectors

Create *random* data following a certain distribution (we will be using this many times in Stats 1)

```
# Randomly choose 3 numbers from a Normal distribution
(my_random_normals <- rnorm(3))
```

```
## [1]  0.7568125 -0.4320825  0.4426344
```

```
# Randomly choose 4 numbers from a Uniform distribution
(my_random_uniforms <- runif(4))
```

```
## [1] 0.5138036 0.2012954 0.2410591 0.6112599
```

The pattern is **r**`distribution` (**r**`unif`, **r**`norm`, **r**`f`, **r**`chisq`)

Doing 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 element
my_numbers + 6
```

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

Try it yourself

Some vectorized functions operate on each value in the vector and return a vector of the same length[3]. Guess the output before running the codes, and verify the results.

```
my_numbers <- 1:6
my_numbers + 2 * my_numbers
my_numbers * my_numbers
my_numbers / my_numbers
```

```
a_vector <- rnorm(200)
sqrt(a_vector) # take the square root of each number
round(a_vector, 2) # round each number to the 2nd decimal place
```

How would you extract all numbers in `a_vector` greater than 1.96?

[3] If not sure, use `?func` to learn more

Warning: Vector recycling

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

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

Warning: Vector recycling

Be careful when operating with vectors. If they're different lengths, the shorter vector starts from it's beginning (`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): longer object len
## multiple of shorter object length
```

```
# 1 + 1,
# 2 + 2,
# 3 + 3,
# 4 + 4,
# 5 + 5,
# 6 + 1 -- '1' is 'recycled'
a
```

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

Binary operators are vectorized

We can do boolean logic with vectors!

```
my_numbers <- 1:6

my_numbers > c(1, 1, 3, 3, pi, pi)
```

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

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

Binary operators are vectorized

We can do boolean logic with vectors!

```r
my_numbers <- 1:6
# Compare my_numbers with 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
```

Accessing Element by Condition (i.e. Logical Vector)

```
x = c(1, 2, 3, 11, 12, 13)
# Choose elements which meet the condition
x < 10
```

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

```
x[x < 10]
```

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

```
# Replace elements which meet the condition with 0
x[x < 10] = 0
x
```

```
## [1]  0  0  0 11 12 13
```

Functions that reduce vectors

Others take a vector and return a summary

- These are used with summarize()

```
a_vector <- c(1, 3, 5, 7, 9, 11, 13)
sum(a_vector)      # add all 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
```

- paste0() is a function that combines character vectors

```
paste0("a", "w", "e", "s", "o", "m", "e")
```

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

Data Types

What is going on here?

```r
a <- "4"
b <- 5
a * b
```

Error in a * b : non-numeric argument to binary operator

What is going on here?

The error we got when we tried `a * b` was because `a` is a character:

```
a <- "4"
b <- 5
a * b # invalid calculation
```

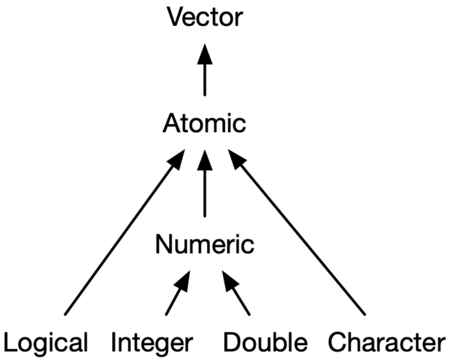
R does not have logic for multiplying character vectors!

Wait what's a character vector?

Data types

R has four primary types of atomic vectors

- these determine how R stores the data (technical)



[4]

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

Data types

Focusing on the primary types, we have:

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

# integer and double, together are called: numeric
type_integer <- 1000
type_double <- 1.0

# character, need to use " " to include the text
type_character <- "abbreviated as chr"
type_character <- "also known as a string"
```

Testing types

```
x <- "1"
typeof(x) # similar result as mode(x) and class(x)
```

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

```
is.integer(x)
```

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

```
is.character(x)
```

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

techincal note: `typeof()` and `mode()` are basically synonyms returning types builtin to R. When programs develop new structures the can assign new `class()`, so class allows for more nuanced results.

Type coercion

The error we got when we tried `a * b` was because `a` is a character:

```
a <- "4"
b <- 5
a * b # invalid calculation
```

We can reassign types on the fly:

```
a <- "4"
b <- 5
as.numeric(a) * b
```

```
## [1] 20
```

What Happens When You Mix Types Inside a Vector?

```r
c(4, "harris")
c(TRUE, 5)
c(FALSE, 100)
c(TRUE, "harris")
```

Character > Numeric > Logical

```r
c(4, "harris")
```

```
## [1] "4"      "harris"
```

```r
c(TRUE, 5)
```

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

```r
c(FALSE, 100)
```

```
## [1]   0 100
```

```r
c(TRUE, "harris")
```

```
## [1] "TRUE"   "harris"
```

Automatic coercion

Logicals are coercible to numeric or character. This can be very useful.

What do you think the following code will return?

```
paste0(FALSE, "?")
mean(c(TRUE, TRUE, FALSE, FALSE, TRUE))
min(c(TRUE, 5, 10))
```

Automatic coercion

```r
# Character > Numeric > Logical
paste0(FALSE, "?")
```

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

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

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

```r
min(c(TRUE, 5, 10))
```

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

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.

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

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

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

NAs are contagious

Think about it `NA` could be **anything** so the output is also *unknown*

```
NA + 4
```

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

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

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

```
mean(c(NA, 3, 4, 5))
```

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

NAs are contagious

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

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

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

```
sum(b, na.rm = TRUE)
```

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

```
mean(b, na.rm = TRUE)
```

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

What do we do we when we want to store different types?

Use lists!

- lets us store arbitrary objects in a single object
- we often use sophisticated objects built on top of lists

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

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] "a"
##
## [[3]]
## [1] TRUE
```

List

We can name the objects in a list for easy reference.

```
my_list <- list(can = TRUE, hold = c(2, 4), anything = "m")
str(my_list)
```

```
## List of 3
##  $ can     : logi TRUE
##  $ hold    : num [1:2] 2 4
##  $ anything: chr "m"
```

Lists

[[and $ pull out a single object from a list by name or location.

```
my_list[[2]]
```

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

```
my_list$anything
```

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

```
typeof(my_list[[2]])
```

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

```
typeof(my_list$anything)
```

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

Lists

We can also subset a [list and retain a list

```
my_list[c(1,3)]
```

```
## $can
## [1] TRUE
##
## $anything
## [1] "m"
```

```
# this is still a list
typeof(my_list[c(1,3)])
```

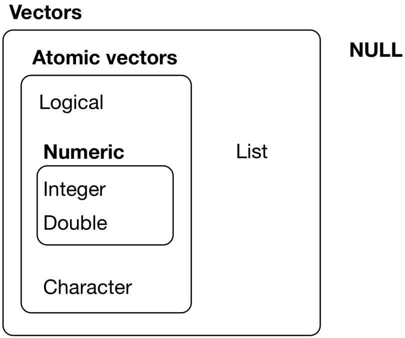
```
## [1] "list"
```

Lists

- lists are still vectors, just not **atomic**

```
is.vector(my_list)
```

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



5

[5] image from https://r4ds.had.co.nz/vectors.html

Empty list creation

To create an empty list of a given size use `vector()`

```
empty_list <- vector("list", 10)
```

Data Frames

Data Frame vs Vector vs List

Vector

Data Frame

List

	country	year	strike.volume	unemployment
1	Australia	1951	296	1.3
2	Australia	1952	397	2.2
3	Australia	1953	360	2.5
4	Australia	1954	3	1.7
5	Australia	1955	326	1.4
6	Australia	1956	352	1.8
7	Australia	1957	195	2.3
8	Australia	1958	133	2.7
9	Australia	1959	109	2.6
10	Australia	1960	208	2.5

- Vector: store elements of the same type
- List: holds elements of different types (e.g. numeric, character, logical)

Columns are vectors

We can create a tibble or data.frame manually

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

```
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
)
```

Could create the same code with `data.frame()`

Ta-da

Take a look at our data set `care_data`:

```
care_data
```

```
## # A tibble: 5 x 4
##      id n_kids child_care_costs random_noise
##   <int>  <dbl>            <dbl>        <dbl>
## 1     1      2             1000         22.9
## 2     2      4             3000        141.
## 3     3      1              300        -26.7
## 4     4      1              300         30.6
## 5     5     NA              500       -332.
```

Rows are lists

(we make use of this idea less often.)

```
bind_rows(
  list(id = 1, n_kids = 2, child_care_costs = 1000),
  list(id = 2, n_kids = 4, child_care_costs = 3000),
  list(id = 5, n_kids = NA, child_care_costs = 500)
)
```

```
## # A tibble: 3 x 3
##      id n_kids child_care_costs
##   <dbl>  <dbl>            <dbl>
## 1     1      2             1000
## 2     2      4             3000
## 3     5     NA              500
```

Detour: Data Analysis via Tidyverse and base R

Tidyverse has become the leading way people clean & manipulate data in R

- These packages make data analysis easier than core base R commands
- Tidyverse commands can be more efficient (less lines of code)

However, you will inevitably run into edge cases where tidyverse commands don't work the way you expect them to, or where you have to reuse/debug codes written in base R ... and hence you'll have to use **base R**

It's good to have a basic foundation on both approaches and then decide which you prefer when conducting data analysis!

Extracting

Base R ways to pull out a column as a vector:

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

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

```r
# base R way (same result as above)
care_data[["n_kids"]]
```

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

Subsetting

Two base R ways to pull out a column as a tibble/data.frame:

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

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

```
care_data[2] # recall n_kids is the second column!
subset(care_data, select = "n_kids")
```

Subsetting and extracting

Notice similarity with lists

- [[and $ for extracting (or pulling)

vs.

- [for subsetting / selecting.

Idea: a data frame is a named list with equal length vectors for each object (i.e. columns)

subsetting [] vs [,]

We saw that using `[]` pulls out columns. (“single index”)

Using `[ , ]` allows us to subset rows and columns. (“double index”)

`data`[get rows , get columns]

Using [with two indices

data[get rows , get columns]

```
care_data[c(1, 3), ]
```

```
## # A tibble: 2 x 4
##      id n_kids child_care_costs random_noise
##   <int>  <dbl>            <dbl>        <dbl>
## 1     1      2             1000         22.9
## 2     3      1              300        -26.7
```

```
care_data[, c(1, 3)]
```

```
## # A tibble: 5 x 2
##      id child_care_costs
##   <int>            <dbl>
## 1     1             1000
## 2     2             3000
## 3     3              300
## 4     4              300
```

Using data[subset rows , *subset columns*]

We can refer to columns by name or index location.

```
care_data[1:2, c("n_kids", "child_care_costs")]
```

```
## # A tibble: 2 x 2
##   n_kids child_care_costs
##    <dbl>            <dbl>
## 1      2             1000
## 2      4             3000
```

Using `data`[subset rows , *subset columns*]

Or even a logical vector. (... this should remind you of vector subsetting!!)

```
care_data[1:2, c(TRUE, FALSE, TRUE, FALSE)]
```

```
## # A tibble: 2 x 2
##      id child_care_costs
##   <int>            <dbl>
## 1     1             1000
## 2     2             3000
```

Using data[*subset rows* , subset columns]

Similarly for rows!

```
care_data[c(1,3), c("id","n_kids")]
```

```
## # A tibble: 2 x 2
##       id n_kids
##    <int>  <dbl>
## 1      1      2
## 2      3      1
```

```
logical_indexing <- c(TRUE, FALSE, TRUE, FALSE, FALSE)
care_data[logical_indexing , c("id","n_kids")]
```

```
## # A tibble: 2 x 2
##       id n_kids
##    <int>  <dbl>
## 1      1      2
## 2      3      1
```

Using data[*subset rows* , subset columns]

More usual usage for logical indexing

```
logical_index <- care_data$id < 3
logical_index
```

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

```
care_data[logical_index, c("id","n_kids")]
```

```
## # A tibble: 2 x 2
##      id n_kids
##   <int>  <dbl>
## 1     1      2
## 2     2      4
```

```
# put the conditional right into the brackets.
care_data[care_data$id < 3 , "id"]
```

Let's get you to try.

First, we need data.

`us_rent_income` is a practice data set that comes tidyverse.

```
library(tidyverse)
head(us_rent_income)
```

```
## # A tibble: 6 x 5
##   GEOID NAME    variable estimate   moe
##   <chr> <chr>   <chr>       <dbl> <dbl>
## 1 01    Alabama income      24476   136
## 2 01    Alabama rent          747     3
## 3 02    Alaska  income      32940   508
## 4 02    Alaska  rent         1200    13
## 5 04    Arizona income      27517   148
## 6 04    Arizona rent          972     4
```

More examples [

Explore the data quickly with `glimpse()`, `nrow()`, etc

How would you use a single bracket `[` . . .

1. to select the state names and variable columns?
2. to get the rows 1, 3, 5, 7 ?
3. to get all the rows about "income".[6]
4. to get the `variable` and `estimate` columns for rows about Illinois?

[6] hint: test if *something* `== "income"`?

subset is a base R function wraps the [

`subset(x, subset, select)` is *almost* equivalent to `x[subset, select]` except subset must be a logical vector!

```
# notice we don't need to refer to the data over and over again!
subset(us_rent_income, variable == "income", c(NAME, variable, estimate
```

```
## # A tibble: 52 x 3
##    NAME                 variable estimate
##    <chr>                <chr>       <dbl>
##  1 Alabama              income      24476
##  2 Alaska               income      32940
##  3 Arizona              income      27517
##  4 Arkansas             income      23789
##  5 California           income      29454
##  6 Colorado             income      32401
##  7 Connecticut          income      35326
##  8 Delaware             income      31560
##  9 District of Columbia income      43198
## 10 Florida              income      25952
## # ... with 42 more rows
```

subset is a base R function wraps the [

Compare:

The following three lines produce identical output.

```
us_rent_income[us_rent_income$variable == "income",
               c("NAME", "variable", "estimate")]

subset(us_rent_income,
       us_rent_income$variable == "income",
       c("NAME", "variable", "estimate"))

subset(us_rent_income, variable == "income", c(NAME, variable, estimate
```

subset is a base R function wraps the [

This reduction in typing allows us to do powerful analysis.

```
subset(us_rent_income,
        variable == "income" &
       (NAME == "Iowa"| NAME == "Alaska"),
       select = c(NAME, variable, estimate))
```

```
## # A tibble: 2 x 3
##   NAME   variable estimate
##   <chr>  <chr>       <dbl>
## 1 Alaska income      32940
## 2 Iowa   income      30002
```

Technical idea: names and environments

When we use `subset()` (and many tidyverse functions), we have access to column `names`.

These are not in the *global environment*, so they are not "found" when using `[`

```
us_rent_income[variable == "income" & (NAME == "Iowa"| NAM
```

Error in `[.tbl_df`(us_rent_income, variable == "income" & (NAME == "Iowa" | : object 'variable' not found

Recap

We discussed how to:

- Create vectors, lists and data frames for various circumstances
- Do vectorized operations and math with vectors
- Subset vectors and lists
- Understand data types and use type coercion when necessary