Basic R

Basic R

Writing in R and Rmarkdown

Chatting with R

Using R is just a chat with the computer.

“Hey, R. What is $1 + 2$?”

```
1 + 2
```

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

Rmarkdown tricks

- To make text **bold**, we add two **s around it.
- To make text *italicized*, we add just one * around it.
- If we need special characters (like * or \$), then we just add a forward “\” in front of them (but not behind).
- Math symbols in your text are process with Latex, just put an “\$” before and after your math. Like this, \$y = x\$ becomes $y = x$.
- Code blocks

To make a code block, press CTRL+ALT+I. Remember you can change the output of a code block by modifying some of its options, like below.

```
banana <- 5

banana + 1
```

```
## [1] 6
```

Variables

The assignment operator

We make variables with the `<-` operator.

```
my_special_var <- 1 + 2

my_special_var
```

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

You can TECHNICALLY use = for assignment too. Never do this.

```
my_other_var = 12

my_other_var + my_special_var
```

```
## [1] 15
```

Numerics

Doubles

Doubles are decimal numbers, like $1.1, 2.2, 3.0$. If I make a number variable without doing anything special, R defaults to a double.

```
a <- 1.1
b <- 2.0

is.double(a)
```

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

```
is.double(b)
```

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

Integers

Integers must have an L after them. That is how R knows that you don't want a double, but instead want a "long-capable integer".

```
c <- 1L
d <- 1

is.integer(c)
```

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

```
is.integer(d)
```

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

Here is a useful cheatsheet for the different numeric operators and how they behave.

Operator	Expression	Result
+	10 + 3	13
-	10 - 3	7
	10 * 3	30
/	10 / 3	3.333
ˆ	10 ˆ 3	1000
%/%	10 %/% 3	3
%%	10 %% 3	1

Why care about the difference?

Almost 99% of the time, this wont matter. But, with big data, integers take up must less memory.

```
my_integers <- seq(from = 1L, to = 1e6L, by = 1L)
my_doubles <- seq(from = 1.0, to = 1e6, by = 1.0)

object.size(my_integers)
```

```
## 4000040 bytes
```

```
object.size(my_doubles)
```

```
## 8000040 bytes
```

Note here that although we are using only whole numbers from 1 to 1 million, the first sequence (`my_integers`) is stored as an integer and the second sequence (`my_doubles`) is stored as a number that may include decimals. This second case needs more space (twice as much) to be allocated in advance, even if we never use those decimal places.

Again, this will almost never matter for most people, most of the time. However, it is good to be aware of for when your datasets get large (i.e., several million cases or more).

Characters

Characters are text symbols and they are made with either "" or ”, either works.

```
a <- 'here is someone\'s text'
b <- "here is more text"

a
```

```
## [1] "here is someone's text"
```

```
b
```

```
## [1] "here is more text"
```

To combine two strings, I use `paste()`.

```
paste(a, b)
```

```
## [1] "here is someone's text here is more text"
```

If I dont want a space, then I used `paste0()`.

```
paste0(a, b)
```

```
## [1] "here is someone's texthere is more text"
```

Booleans

These are True and False values. You make them with the symbols `T` or `TRUE` and `F` or `FALSE`.

```
x <- T
y <- F
```

To compare them, we can use three operators.

- `&` is "and"
- `|` is "or"
- `!` is "not" (just give me the opposite of whatever is after me)

```
x & y # false
```

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

```
x | y # true
```

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

```
x & !y # true
```

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

We can also have nested equations

```
z <- F

x & !(y | z) # true
```

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

We can also compare numbers.

```
a <- 1
b <- 2
```

```
a < 1
```

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

```
a <= 1
```

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

```
a == 1
```

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

If I want to compare multiple numbers, I need to do it seperately.

```
(a > 1) | (b > 1)
```

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

Remember that booleans are ultimately numeric values underneath.

```
d <- T
k <- F
u <- 5

d*u
```

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

```
d*k
```

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

```
as.numeric(d)
```

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

```
as.numeric(k)
```

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

Special types

`NA` - missing

```
is.na(NA)
```

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

`NaN` - you did math wrong

```
0/0
```

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

`Inf` - infinity

```
-5/0
```

```
## [1] -Inf
```

Vectors

R is built is on vectors. Vectors are collections of a bunch of values of the same type.

```
my_vec <- c(1, 5, 3, 7)

my_vec
```

```
## [1] 1 5 3 7
```

If I try to put different types together, they go to the most primitive type (usually a character string).

```
my_other_vec <- c(22, 'orange', T)

my_other_vec
```

```
## [1] "22"     "orange" "TRUE"
```

```
my_third_vec <- c(T, F, 35)

my_third_vec
```

```
## [1]  1  0 35
```

We can also missing values.

```
my_fourth_vec <- c(1, 4, 5, NA)

my_fourth_vec
```

```
## [1]  1  4  5 NA
```

```
is.na(my_fourth_vec)
```

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

If I want to combine two vectors…

```
a <- c(1, 2, 3)
b <- c(3, 5, 7)

c(a, b)
```

```
## [1] 1 2 3 3 5 7
```

A brief example of matrices

```
matrix(
  data = c(a, b),
  nrow = 2,
  byrow = T)
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    3    5    7
```

Sometimes I want special vectors, direct sequences of numbers. There are two ways to do this. If all I want is a integer sequence (made of doubles), then I use the "`<first number>:<last number>`".

```
1:5
```

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

```
5:1
```

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

Other times, I need to count by something other than one, so I use `seq(from = <start>, to = <end>, by = <number to count by>)`

```
seq(from = 1, to = 7, by = 1.3)
```

```
## [1] 1.0 2.3 3.6 4.9 6.2
```

Hint: for brevity, I can leave off function parameter names, **as long as I enter them in order**

```
seq(1, 7, by = 1.3)
```

```
## [1] 1.0 2.3 3.6 4.9 6.2
```

If I add a constant to a vector, then they all go up by that constant.

```
1:5 / 3
```

```
## [1] 0.3333333 0.6666667 1.0000000 1.3333333 1.6666667
```

I can do math with equal-length sequences too.

```
1:5 - seq(1, 4, by = .7)
```

```
## [1] 0.0 0.3 0.6 0.9 1.2
```

But they **must** be equal lengths.

```
1:5 / 1:4
```

```
## Warning in 1:5/1:4: longer object length is not a multiple of shorter
## object length
```

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

To access the elements of a vector, I put a number OR booleans in brackets `[]`.

```
my_vec <- c('apple', 'orange', 'banana', 'pair')

my_vec[2]
```

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

```
my_vec[2:4]
```

```
## [1] "orange" "banana" "pair"
```

```
my_vec[c(3, 2, 1, 4)]
```

```
## [1] "banana" "orange" "apple"  "pair"
```

I can also use bools.

```
my_other_vec <- c(1, 4, 6, 7, 9, 3, 9)

my_other_vec < 5
```

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

```
my_other_vec[my_other_vec < 5]
```

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

I can also use functions that return values to access vectors, if I am creative…

```
my_other_vec[max(my_other_vec) == my_other_vec]
```

```
## [1] 9 9
```

R also has special vectors that are pre-loaded. The most commonly used are `letters` and `LETTERS`, which return the lower-case letters and uppercase letters of the English alphabet, respectively.

```
vec <- c(1, 3, 4, 5, 3, 2, NA)

mean(vec, na.rm = T)
```

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

Lists

Lists are a kind of vector that we make using the `list()` function, rather than the `c()` function. They have a few special properties.

Lists

Lists are special vectors that can hold multiple types of elements.

They can even hold other vectors.

Dataframes

Construction

Dataframes are spreadsheets. Under the hood of R, they are just lists of vectors, where all the vectors are required to be the same length. To make one, you can call the `data.frame()` function and put your vectors inside.

```
heights <- c(60, 65, 71, 72, 64)
sexes <- c('female', 'female', 'male', 'male', 'female')
shoes <- c('Adidas', 'Nike', 'Nike', 'Salvatore Ferragamo', 'Reebok')

df <- data.frame(height = heights, sex = sexes, shoes = shoes)

df
```

```
##   height    sex               shoes
## 1     60 female              Adidas
## 2     65 female                Nike
## 3     71   male                Nike
## 4     72   male Salvatore Ferragamo
## 5     64 female              Reebok
```

Built-in dataframes

R has numerous built-in datasets that are ideal for demonstration purposes. We can get access to them using the `data()` command. This will load the data into our session, so we can then look at it.

```
data('mtcars')

mtcars
```

```
##                      mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4           21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag       21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710          22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive      21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
## Valiant             18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
## Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
## Merc 240D           24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## Merc 230            22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
## Merc 280            19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
## Merc 280C           17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
## Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
## Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
## Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
## Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
## Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
## Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
## Fiat 128            32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
## Honda Civic         30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
## Toyota Corolla      33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
## Toyota Corona       21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
## Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
## AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
## Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
## Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
## Fiat X1-9           27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
## Porsche 914-2       26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
## Lotus Europa        30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
## Ford Pantera L      15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
## Ferrari Dino        19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
## Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
## Volvo 142E          21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

Some datasets do not come in the form of a dataframe right away, but they can be converted into one using the `as.data.frame()` function.

```r
data(Seatbelts)

is.data.frame(Seatbelts)
```

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

```r
seatbelts_df <- as.data.frame(Seatbelts)

is.data.frame(seatbelts_df)
```

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

Functions

Packages