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
^	10 ^ 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.

```
object.size(1L:1e6)
```

```
## 4000040 bytes
```

XXXX <- Ian, figure out why this isnt work.

```
pryr::object_size(1L:1e6L)
```

```
## 4 MB
```

```
pryr::object_size(seq(1.0, 1e6))
```

```
## 4 MB
```

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.

```r
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).

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

my_other_vec
```

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

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

my_third_vec
```

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

We can also missing values.

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

my_fourth_vec
```

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

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

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

If I want to combine two vectors...

```r
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

```r
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 `[]`.

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

my_vec[2]
```

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

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

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

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

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

I can also use bools.

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

my_other_vec < 5
```

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

```r
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...

```r
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.

```r
letters
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q"
## [18] "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

```r
LETTERS
```

```
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q"
## [18] "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
```

```r
vec3 <- c(5, 4, 3, 7, 8, 2, 1)

vec3[vec3 %% 2 == 0]
```

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

Lists

Dataframes

Construction Accessing elements inside the dataframe - `$` - Rows with numbers - Rows with boolean tests - Columns with numbers - Columns with names - Making new columns **Basic data modification**

Basic functions

What are functions? Common functions `rnorm()` `mean()` `sd()` **What about missing data? How can I make my own?**

Packages

Base R and the need for packages

Package installation

Package loading

Calling functions without loading a package