Logic and types in R

Lecture 02

Dr. Colin Rundel

In R (almost) everything is a vector

Vectors

The fundamental building block of data in R are vectors (collections of related values, objects, data structures, etc).

R has two types of vectors:

- **atomic** vectors (*vectors*)

  - homogeneous collections of the *same* type (e.g. all `true`/`false` values, all numbers, or all character strings).

- **generic** vectors (*lists*)

  - heterogeneous collections of *any* type of R object, even other lists (meaning they can have a hierarchical/tree-like structure).

Atomic Vectors

Atomic Vectors

R has six atomic vector types, we can check the type of any object in R using the `typeof()` function

typeof()	mode()
logical	logical
double	numeric
integer	numeric
character	character
complex	complex
raw	raw

Mode is a higher level abstraction, we will discuss this in detail a bit later.

logical - boolean values (TRUE and FALSE)

```
1  typeof(TRUE)
```

```
[1] "logical"
```

```
1  typeof(FALSE)
```

```
[1] "logical"
```

```
1  mode(TRUE)
```

```
[1] "logical"
```

```
1  mode(FALSE)
```

```
[1] "logical"
```

R will let you use T and F as shortcuts to TRUE and FALSE, this is a bad practice as these values are actually *global variables* that can be overwritten.

```
1  T
```

```
[1] TRUE
```

```
1  T = FALSE
2  T
```

```
[1] FALSE
```

character - text strings

Either single or double quotes are fine, opening and closing quote must match.

```
1 typeof("hello")
```

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

```
1 typeof('world')
```

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

```
1 mode("hello")
```

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

```
1 mode('world')
```

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

Quote characters can be included by escaping or using a non-matching quote.

```
1 "abc'123"
```

```
[1] "abc'123"
```

```
1 'abc"123'
```

```
[1] "abc\"123"
```

```
1 "abc\"123"
```

```
[1] "abc\"123"
```

```
1 'abc\'123'
```

```
[1] "abc'123"
```

RStudio's syntax highlighting is helpful here to indicate where it thinks a string begins and ends.

Numeric types

`double` - floating point values (these are the default numerical type)

```
1  typeof(1.33)
```

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

```
1  typeof(7)
```

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

```
1  mode(1.33)
```

```
[1] "numeric"
```

```
1  mode(7)
```

```
[1] "numeric"
```

`integer` - integer values (literals are indicated with an `L` suffix)

```
1  typeof( 7L )
```

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

```
1  typeof( 1:3 )
```

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

```
1  mode( 7L )
```

```
[1] "numeric"
```

```
1  mode( 1:3 )
```

```
[1] "numeric"
```

Concatenation

Atomic vectors can be grown (combined) using the combine `c()` function.

```r
1  c(1, 2, 3)
```

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

```r
1  c("Hello", "World!")
```

```
[1] "Hello"  "World!"
```

```r
1  c(1, 1:10)
```

```
 [1]  1  1  2  3  4  5  6  7  8  9 10
```

```r
1  c(1,c(2, c(3)))
```

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

Note – atomic vectors are inherently flat.

Inspecting types

- `typeof(x)` – returns a character vector (length 1) of the *type* of object `x`.

- `mode(x)` – returns a character vector (length 1) of the *mode* of object `x`.

```
1  typeof(1)
```

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

```
1  typeof(1L)
```

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

```
1  typeof("A")
```

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

```
1  typeof(TRUE)
```

```
[1] "logical"
```

```
1  mode(1)
```

```
[1] "numeric"
```

```
1  mode(1L)
```

```
[1] "numeric"
```

```
1  mode("A")
```

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

```
1  mode(TRUE)
```

```
[1] "logical"
```

Type predicates

- `is.logical(x)` - returns TRUE if x has *type* `logical`.

- `is.character(x)` - returns TRUE if x has *type* `character`.

- `is.double(x)` - returns TRUE if x has *type* `double`.

- `is.integer(x)` - returns TRUE if x has *type* `integer`.

- `is.numeric(x)` - returns TRUE if x has *mode* `numeric`.

```
1  is.integer(1)
```

```
[1] FALSE
```

```
1  is.integer(1L)
```

```
[1] TRUE
```

```
1  is.integer(3:7)
```

```
[1] TRUE
```

```
1  is.double(1)
```

```
[1] TRUE
```

```
1  is.double(1L)
```

```
[1] FALSE
```

```
1  is.double(3:8)
```

```
[1] FALSE
```

```
1  is.numeric(1)
```

```
[1] TRUE
```

```
1  is.numeric(1L)
```

```
[1] TRUE
```

```
1  is.numeric(3:7)
```

```
[1] TRUE
```

Other useful predicates

- `is.atomic(x)` - returns TRUE if `x` is an *atomic vector*.

- `is.list(x)` - returns TRUE if `x` is a *list* (generic vector).

- `is.vector(x)` - returns TRUE if `x` is either an *atomic* or *generic* vector.

```
1  is.atomic(c(1,2,3))
```

```
[1] TRUE
```

```
1  is.list(c(1,2,3))
```

```
[1] FALSE
```

```
1  is.vector(c(1,2,3))
```

```
[1] TRUE
```

```
1  is.atomic(list(1,2,3))
```

```
[1] FALSE
```

```
1  is.list(list(1,2,3))
```

```
[1] TRUE
```

```
1  is.vector(list(1,2,3))
```

```
[1] TRUE
```

Type Coercion

R is a dynamically typed language – it will automatically convert between most types without raising warnings or errors. Keep in mind that atomic vectors must always contain values of the same type.

```r
1  c(1, "Hello")
```

```
[1] "1"     "Hello"
```

```r
1  c(FALSE, 3L)
```

```
[1] 0 3
```

```r
1  c(1.2, 3L)
```

```
[1] 1.2 3.0
```

```r
1  c(FALSE, "Hello")
```

```
[1] "FALSE" "Hello"
```

Operator coercion

Builtin operators and functions (e.g. +, &, log(), etc.) will generally attempt to coerce values to an appropriate type for the given operation

```
1  3.1+1L
```

```
[1] 4.1
```

```
1  5 + FALSE
```

```
[1] 5
```

```
1  TRUE & FALSE
```

```
[1] FALSE
```

```
1  TRUE & 7
```

```
[1] TRUE
```

```
1  log(1)
```

```
[1] 0
```

```
1  log(TRUE)
```

```
[1] 0
```

```
1  TRUE | FALSE
```

```
[1] TRUE
```

```
1  FALSE | !5
```

```
[1] FALSE
```

Explicit Coercion

Most of the `is` functions we just saw have an `as` variant which can be used for *explicit* coercion.

```
1  as.logical(5.2)
```

```
[1] TRUE
```

```
1  as.character(TRUE)
```

```
[1] "TRUE"
```

```
1  as.integer(pi)
```

```
[1] 3
```

```
1  as.numeric(FALSE)
```

```
[1] 0
```

```
1  as.double("7.2")
```

```
[1] 7.2
```

```
1  as.double("one")
```

```
[1] NA
```

Missing Values

Missing Values

R uses NA to represent missing values in its data structures, what may not be obvious is that there are different NAs for different atomic types.

```
1  typeof(NA)
```

```
[1] "logical"
```

```
1  typeof(NA+1)
```

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

```
1  typeof(NA+1L)
```

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

```
1  typeof(c(NA,""))
```

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

```
1  typeof(NA_character_)
```

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

```
1  typeof(NA_real_)
```

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

```
1  typeof(NA_integer_)
```

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

```
1  typeof(NA_complex_)
```

```
[1] "complex"
```

NA "stickiness"

Because NAs represent missing values it makes sense that any calculation using them should also be missing.

```
1  1 + NA
```

```
[1] NA
```

```
1  1 / NA
```

```
[1] NA
```

```
1  NA * 5
```

```
[1] NA
```

```
1  sqrt(NA)
```

```
[1] NA
```

```
1  3^NA
```

```
[1] NA
```

```
1  sum(c(1, 2, 3, NA))
```

```
[1] NA
```

Summarizing functions (e.g. sum(), mean(), sd(), etc.) will often have a na.rm argument which will allow you to *drop* missing values.

```
1  sum(c(1, 2, 3, NA), na.rm = TRUE)
```

```
[1] 6
```

```
1  mean(c(1, 2, 3, NA), na.rm = TRUE)
```

```
[1] 2
```

NAs are not always sticky

A useful mental model for NAs is to consider them as a unknown value that could take any of the possible values for a type.

For numbers or characters this isn't very helpful, but for a logical value we know that the value must either be TRUE or FALSE and we can use that when deciding what value to return.

```
1  TRUE & NA
```

```
[1] NA
```

```
1  FALSE & NA
```

```
[1] FALSE
```

```
1  TRUE | NA
```

```
[1] TRUE
```

```
1  FALSE | NA
```

```
[1] NA
```

Other Special values (double)

These are defined as part of the IEEE floating point standard (not unique to R)

- NaN - Not a number

- Inf - Positive infinity

- -Inf - Negative infinity

```
1 pi / 0
```

[1] Inf

```
1 0 / 0
```

[1] NaN

```
1 1/0 + 1/0
```

[1] Inf

```
1 1/0 - 1/0
```

[1] NaN

```
1 NaN / NA
```

[1] NA

```
1 NaN * NA
```

[1] NA

Testing for `Inf` and `NaN`

`NaN` and `Inf` don't have the same testing issues that `NA`s do, but there are still convenience functions for testing for these types of values

```
1  is.finite(Inf)
```

```
[1] FALSE
```

```
1  is.infinite(-Inf)
```

```
[1] TRUE
```

```
1  is.nan(Inf)
```

```
[1] FALSE
```

```
1  is.nan(-Inf)
```

```
[1] FALSE
```

```
1  Inf > 1
```

```
[1] TRUE
```

```
1  -Inf > 1
```

```
[1] FALSE
```

```
1  is.finite(NaN)
```

```
[1] FALSE
```

```
1  is.infinite(NaN)
```

```
[1] FALSE
```

```
1  is.nan(NaN)
```

```
[1] TRUE
```

```
1  is.finite(NA)
```

```
[1] FALSE
```

```
1  is.infinite(NA)
```

```
[1] FALSE
```

```
1  is.nan(NA)
```

```
[1] FALSE
```

Coercion for infinity and NaN

First remember that `Inf`, `-Inf`, and `NaN` are doubles, however their coercion behavior is not the same as other doubles

```
1  as.integer(Inf)
```

```
[1] NA
```

```
1  as.integer(NaN)
```

```
[1] NA
```

```
1  as.logical(Inf)
```

```
[1] TRUE
```

```
1  as.logical(-Inf)
```

```
[1] TRUE
```

```
1  as.logical(NaN)
```

```
[1] NA
```

```
1  as.character(Inf)
```

```
[1] "Inf"
```

```
1  as.character(-Inf)
```

```
[1] "-Inf"
```

```
1  as.character(NaN)
```

```
[1] "NaN"
```

Exercise 1

Part 1

What is the type of the following vectors? Explain why they have that type.

- `c(1, NA+1L, "C")`

- `c(1L / 0, NA)`

- `c(1:3, 5)`

- `c(3L, NaN+1L)`

- `c(NA, TRUE)`

Part 2

Considering only the four (common) data types, what is R's implicit type conversion hierarchy (from highest priority to lowest priority)?

Hint – think about the pairwise interactions between types.

05:00

Conditionals & Control Flow

Logical (boolean) operators

Operator	Operation	Vectorized?
x | y	or	Yes
x & y	and	Yes
!x	not	Yes
x || y	or	No
x && y	and	No
xor(x, y)	exclusive or	Yes

Vectorized?

```
1  x = c(TRUE,FALSE,TRUE)
2  y = c(FALSE,TRUE,TRUE)
```

```
1  x | y
```

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

```
1  x & y
```

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

```
1  x || y
```

```
Error in x || y: 'length = 3' in
coercion to 'logical(1)'
```

```
1  x && y
```

```
Error in x && y: 'length = 3' in
coercion to 'logical(1)'
```

Note previously (before R 4.3) both || and && only use the *first* value in the vector, all other values are ignored,

Vectorization and math

Almost all of the basic mathematical operations (and many other functions) in R are vectorized.

```
1 c(1, 2, 3) + c(3, 2, 1)
```

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

```
1 c(1, 2, 3) / c(3, 2, 1)
```

```
[1] 0.3333333 1.0000000 3.0000000
```

```
1 log(c(1, 3, 0))
```

```
[1] 0.000000 1.098612     -Inf
```

```
1 sin(c(1, 2, 3))
```

```
[1] 0.8414710 0.9092974 0.1411200
```

Length coercion (aka recycling)

If the lengths of the vector do not match, then the shorter vector has its values recycled to match the length of the longer vector.

```
1  x = c(TRUE, FALSE, TRUE)
2  y = c(TRUE)
3  z = c(FALSE, TRUE)
```

```
1  x | y
```

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

```
1  x & y
```

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

```
1  y | z
```

```
[1] TRUE TRUE
```

```
1  y & z
```

```
[1] FALSE  TRUE
```

```
1  x | z
```

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

Length coercion and math

The same length coercion rules apply for most basic mathematical operators,

```
1  x = c(1, 2, 3)
2  y = c(5, 4)
3  z = 10L
```

```
1  x + x
```

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

```
1  x + z
```

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

```
1  x %% y
```

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

```
1  y / z
```

```
[1] 0.5 0.4
```

```
1  log(x)+z
```

```
[1] 10.00000 10.69315 11.09861
```

Comparison operators

Operator	Comparison	Vectorized?
x < y	less than	Yes
x > y	greater than	Yes
x <= y	less than or equal to	Yes
x >= y	greater than or equal to	Yes
x != y	not equal to	Yes
x == y	equal to	Yes
x %in% y	contains	Yes (over x)

over x here means the returned value will have the length of x regardless of the length of y

Comparisons

```r
1 x = c("A","B","C")
2 y = c("A")
```

```r
1 x == y
```

```r
1 x %in% y
```

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

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

```r
1 x != y
```

```r
1 y %in% x
```

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

```
[1] TRUE
```

Type coercion also applies for comparison opperators which can result in *interesting behavior*

```r
1 TRUE == "TRUE"
```

```r
1 TRUE == 1
```

```
[1] TRUE
```

```
[1] TRUE
```

```r
1 FALSE == 1
```

```r
1 TRUE == 5
```

```
[1] FALSE
```

```
[1] FALSE
```

> & < with characters

While maybe somewhat unexpected, these comparison operators can be used character values.

```
1  "A" < "B"
```

[1] TRUE

```
1  "A" > "B"
```

[1] FALSE

```
1  "A" < "a"
```

[1] FALSE

```
1  "a" > "!"
```

[1] TRUE

```
1  "Good" < "Goodbye"
```

[1] TRUE

```
1  c("Alice", "Bob", "Carol") <= "B"
```

[1] TRUE FALSE FALSE

Note – to better understand how this works / the ordering used take a look at ASCII code

Conditional Control Flow

Conditional execution of code blocks is achieved via `if` statements.

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

```
1  if (3 %in% x) {
2    print("Contains 3!")
3  }
```

```
[1] "Contains 3!"
```

```
1  if (1 %in% x)
2    print("Contains 1!")
```

```
[1] "Contains 1!"
```

```
1  if (5 %in% x) {
2    print("Contains 5!")
3  }
```

```
1  if (5 %in% x) {
2    print("Contains 5!")
3  } else {
4    print("Does not contain 5!")
5  }
```

```
[1] "Does not contain 5!"
```

`if` is not vectorized

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

```
1  if (x == 1)
2    print("x is 1!")
```

Error in if (x == 1) print("x is 1!"): the condition has length > 1

```
1  if (x == 3)
2    print("x is 3!")
```

Error in if (x == 3) print("x is 3!"): the condition has length > 1

Note that the behavior seen above (thrown errors) is new in R 4.2, previous versions will only throw warnings

Collapsing logical vectors

There are a couple of helper functions for collapsing a logical vector down to a single value: any, all

```
1  x = c(3,4,1)
```

```
1  x >= 2
```

```
1  x <= 4
```

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

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

```
1  any(x >= 2)
```

```
1  any(x <= 4)
```

```
[1] TRUE
```

```
[1] TRUE
```

```
1  all(x >= 2)
```

```
1  all(x <= 4)
```

```
[1] FALSE
```

```
[1] TRUE
```

```
1  if (any(x == 3))
2    print("x contains 3!")
```

```
[1] "x contains 3!"
```

else if and else

```
1  x = 3
2
3  if (x < 0) {
4    "x is negative"
5  } else if (x > 0) {
6    "x is positive"
7  } else {
8    "x is zero"
9  }
```

```
[1] "x is positive"
```

```
1  x = 0
2
3  if (x < 0) {
4    "x is negative"
5  } else if (x > 0) {
6    "x is positive"
7  } else {
8    "x is zero"
9  }
```

```
[1] "x is zero"
```

if and return

R's `if` conditional statements return a value (invisibly), the two following implementations are equivalent.

```
1  x = 5
```

```
1  s = if (x %% 2 == 0) {
2    x / 2
3  } else {
4    3*x + 1
5  }
```

```
1  s
```

[1] 16

```
1  x = 5
```

```
1  if (x %% 2 == 0) {
2    s = x / 2
3  } else {
4    s = 3*x + 1
5  }
```

```
1  s
```

[1] 16

Notice that conditional expressions are evaluated in the parent scope.

Exercise 2

Take a look at the following code below on the left, without running it in R what do you expect the outcome will be for each call on the right?

```r
f = function(x) {
  # Check small prime
  if (x > 10 || x < -10) {
    stop("Input too big")
  } else if (x %in% c(2, 3, 5, 7)) {
    cat("Input is prime!\n")
  } else if (x %% 2 == 0) {
    cat("Input is even!\n")
  } else if (x %% 2 == 1) {
    cat("Input is odd!\n")
  }
}
```

```r
f(1)
f(3)
f(8)
f(-1)
f(-3)
f(1:2)
f("0")
f("3")
f("zero")
```

More on functions next time

05:00

Conditionals and missing values

NAs can be particularly problematic for control flow,

```r
1  if (2 != NA) {
2    "Here"
3  }
```

Error in if (2 != NA) {: missing value where
TRUE/FALSE needed

```r
1  2 != NA
```

```
[1] NA
```

```r
1  if (all(c(1,2,NA,4) >= 1)) {
2    "There"
3  }
```

Error in if (all(c(1, 2, NA, 4) >= 1)) {: missing
value where TRUE/FALSE needed

```r
1  all(c(1,2,NA,4) >= 1)
```

```
[1] NA
```

```r
1  if (any(c(1,2,NA,4) >= 1)) {
2    "There"
3  }
```

```
[1] "There"
```

```r
1  any(c(1,2,NA,4) >= 1)
```

```
[1] TRUE
```

Testing for NA

To explicitly test if a value is missing it is necessary to use `is.na` (often along with `any` or `all`).

```r
1  NA == NA
```

```
[1] NA
```

```r
1  is.na(NA)
```

```
[1] TRUE
```

```r
1  is.na(1)
```

```
[1] FALSE
```

```r
1  is.na(c(1,2,3,NA))
```

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

```r
1  any(is.na(c(1,2,3,NA)))
```

```
[1] TRUE
```

```r
1  all(is.na(c(1,2,3,NA)))
```

```
[1] FALSE
```