Logic and types in R

Lecture 01

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, etc).

R has two types of vectors (that everything is built on):

- 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.

There are additional types in R, e.g. `list`, `closure`, `environment`, etc. We will see these in the next couple of weeks. Check `?typeof` for more information.

`logical` - boolean values (`TRUE` and `FALSE`)

```r
typeof(TRUE)
```

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

```r
typeof(FALSE)
```

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

```r
mode(TRUE)
```

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

```r
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.

R Code Start Over Run Code

```r
T
```

`character` - text strings

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

```r
typeof("hello")
```

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

```r
typeof('world')
```

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

```r
mode("hello")
```

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

```r
mode('world')
```

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

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

```r
"abc'123"
```

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

```r
'abc"123'
```

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

```r
"abc\"123"
```

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

```r
'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)

```r
typeof(1.33)
```

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

```r
mode(1.33)
```

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

```r
typeof(7)
```

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

```r
mode(7)
```

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

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

```r
typeof( 7L )
```

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

```r
mode( 7L )
```

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

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

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

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

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

Combining / Concatenation

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

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

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

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

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

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

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

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

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

Note - atomic vectors are inherently flat / 1d.

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`.

```r
typeof(1)
```

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

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

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

```r
typeof("A")
```

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

```r
typeof(TRUE)
```

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

```r
mode(1)
```

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

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

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

```r
mode("A")
```

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

```r
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`.

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

```
[1] FALSE
```

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

```
[1] TRUE
```

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

```
[1] TRUE
```

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

```
[1] TRUE
```

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

```
[1] FALSE
```

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

```
[1] FALSE
```

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

```
[1] TRUE
```

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

```
[1] TRUE
```

```r
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.

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

```
[1] TRUE
```

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

```
[1] FALSE
```

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

```
[1] FALSE
```

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

```
[1] TRUE
```

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

```
[1] TRUE
```

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

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

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

```
[1] 0 3
```

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

```
[1] 1.2 3.0
```

```r
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 (numeric for math, logical for logical, etc.)

```r
3.1+1L
```

```
[1] 4.1
```

```r
5 + FALSE
```

```
[1] 5
```

```r
TRUE & FALSE
```

```
[1] FALSE
```

```r
TRUE & 7
```

```
[1] TRUE
```

```r
log(1)
```

```
[1] 0
```

```r
log(TRUE)
```

```
[1] 0
```

```r
TRUE | FALSE
```

```
[1] TRUE
```

```r
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.

```r
as.logical(5.2)
```

```
[1] TRUE
```

```r
as.character(TRUE)
```

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

```r
as.integer(pi)
```

```
[1] 3
```

```r
as.numeric(FALSE)
```

```
[1] 0
```

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

```
[1] 7.2
```

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

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

```
[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 `NA`s for the different atomic types.

```r
typeof(NA)
```

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

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

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

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

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

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

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

```r
typeof(NA_character_)
```

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

```r
typeof(NA_real_)
```

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

```r
typeof(NA_integer_)
```

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

```r
typeof(NA_complex_)
```

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

This should make sense as `NA` values can appear along over values in atomic vectors.

NA stickiness

Because `NA`s represent missing values it makes sense that most calculations using them will also be missing.

```r
1 + NA
```

```
[1] NA
```

```r
sqrt(NA)
```

```
[1] NA
```

```r
1 / NA
```

```
[1] NA
```

```r
3^NA
```

```
[1] NA
```

```r
NA * 5
```

```
[1] NA
```

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

```
[1] NA
```

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

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

```
[1] 6
```

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

```
[1] 2
```

NAs are not always sticky

A useful mental model for `NA`s 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.

```r
TRUE & NA
```

```
[1] NA
```

```r
FALSE & NA
```

```
[1] FALSE
```

```r
TRUE | NA
```

```
[1] TRUE
```

```r
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

```r
pi / 0
```

```
[1] Inf
```

```r
0 / 0
```

```
[1] NaN
```

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

```
[1] Inf
```

```r
Inf - Inf
```

```
[1] NaN
```

```r
NaN / NA
```

```
[1] NA
```

```r
NaN * NA
```

```
[1] NA
```

Testing for `Inf` and `NaN`

`NaN` and `Inf` there are convenience functions for testing for these types of values

```r
is.finite(Inf)
```

```
[1] FALSE
```

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

```
[1] TRUE
```

```r
is.nan(Inf)
```

```
[1] FALSE
```

```r
Inf > 1
```

```
[1] TRUE
```

```r
is.finite(NaN)
```

```
[1] FALSE
```

```r
is.infinite(NaN)
```

```
[1] FALSE
```

```r
is.nan(NaN)
```

```
[1] TRUE
```

```r
-Inf > 1
```

```
[1] FALSE
```

```r
is.finite(NA)
```

```
[1] FALSE
```

```r
is.infinite(NA)
```

```
[1] FALSE
```

```r
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

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

```
Warning: NAs introduced by coercion to integer range
[1] NA
```

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

```
[1] NA
```

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

```
[1] TRUE
```

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

```
[1] TRUE
```

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

```
[1] NA
```

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

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

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

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

```
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?

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

```
x | y
```

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

```
x & y
```

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

```
x || y
```

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

```
x && y
```

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

`&` and `|` are almost always going to be the right choice, the only time we use `&&` or `||` is when you need to take advantage of short-circuit evaluation.

Note previously (before R 4.3) both `||` and `&&` only used the *first* value in the vector, all other values are ignored and there was no warning about the ignored values.

Vectorization and math

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

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

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

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

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

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

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

```r
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.

```r
x = c(TRUE, FALSE, TRUE)
y = c(TRUE)
z = c(FALSE, TRUE)
```

```r
x | y
```

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

```r
y | z
```

```
[1] TRUE TRUE
```

```r
x & y
```

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

```r
y & z
```

```
[1] FALSE  TRUE
```

```r
x | z
```

```
Warning in x | z: longer object length is not a multiple of
shorter object
length
```

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

Length coercion and math

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

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

```r
x + x
```

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

```r
y / z
```

```
[1] 0.5 0.4
```

```r
x + z
```

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

```r
log(x)+z
```

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

```r
x %% y
```

```
Warning in x%%y: longer object length is not a multiple of
shorter object
length
```

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

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` means the returned value will have the length of `x` regardless of the length of `y`

Comparisons

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

```r
x == y
```

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

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

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

```r
x != y
```

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

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

```
[1] TRUE
```

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

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

```
[1] TRUE
```

```r
TRUE == 1
```

```
[1] TRUE
```

```r
FALSE == 1
```

```
[1] FALSE
```

```r
TRUE == 5
```

```
[1] FALSE
```

> & < with characters

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

```r
"A" < "B"
```

```
[1] TRUE
```

```r
"A" > "B"
```

```
[1] FALSE
```

```r
"A" < "a"
```

```
[1] FALSE
```

```r
"a" > "!"
```

```
[1] TRUE
```

```r
"Good" < "Goodbye"
```

```
[1] TRUE
```

```r
c("Alice", "Bob", "Carol") <
```

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

Note - to better understand how this works, i.e. the ordering used, see ASCII code

Conditional Control Flow

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

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

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

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

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

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

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

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

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

`if` is not vectorized

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

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

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

```r
if (x == 3)
  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 (using only the first value in the condition vector).

Collapsing logical vectors

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

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

```r
x >= 2
```

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

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

```
[1] TRUE
```

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

```
[1] FALSE
```

```r
x <= 4
```

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

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

```
[1] TRUE
```

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

```
[1] TRUE
```

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

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

else if and else

```r
x = 3

if (x < 0) {
  "x is negative"
} else if (x > 0) {
  "x is positive"
} else {
  "x is zero"
}
```

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

```r
x = 0

if (x < 0) {
  "x is negative"
} else if (x > 0) {
  "x is positive"
} else {
  "x is zero"
}
```

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

`if` return values

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

```r
x = 5
```

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

```r
s
```

```
[1] 16
```

```r
x = 5
```

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

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

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

```r
2 != NA
```

```
[1] NA
```

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

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

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

```
[1] NA
```

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

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

```r
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
NA == NA
```

```
[1] NA
```

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

```
[1] TRUE
```

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

```
[1] FALSE
```

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

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

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

```
[1] TRUE
```

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

```
[1] FALSE
```