

SM-2302 Software for Mathematicians

R1: Logic and Types in R

Dr. Haziq Jamil

 $\label{eq:mathematical Sciences} \mbox{Mathematical Sciences, Faculty of Science, UBD}$

https://haziqj.ml

Semester I 2023/24

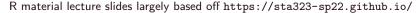
Introduction

In R (almost) everything is a vector

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

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)

```
typeof (TRUE)
                                                  mode (TRUE)
                                                  ## [1] "logical"
## [1] "logical"
                                                  mode(FALSE)
typeof (FALSE)
                                                  ## [1] "logical"
## [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] TRUE T <- FALSE

[1] FALSE

character - text strings

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

```
typeof("hello")
                                                mode("hello")
                                                ## [1] "character"
## [1] "character"
                                                mode('world')
typeof('world')
                                                ## [1] "character"
## [1] "character"
Quote characters can be included by escaping or using a non-matching quote.
```

[1] "abc'123"

"abc '123"

'abc"123'

[1] "abc\"123"



Numeric types

5 / 44

t ger"

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

```
typeof (1.33)
                                                 mode(1.33)
                                                 ## [1] "numeric"
## [1] "double"
                                                 mode(7)
typeof(7)
                                                 ## [1] "numeric"
## [1] "double"
integer - integer values (literals are indicated with an L suffix)
                                                 mode(7L)
typeof(7L)
                                                 ## [1] "numeric"
## [1] "integer"
                                                 mode(1:3)
typeof( 1:3 )
```

[1] "numeric"

Universiti Brunei

Concatenation

Atomic vectors can be grown (combined) using the concatenate 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.

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.

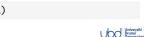
• str(x): compactly display	the internal <i>str</i> ucture of object	x.
typeof(1)	mode(1)	str(1)



##	[1]	"double"	##	[1]	"numeric"	

ypeof(1L)	mode(1L)





[1] "integer"

[1] "character"

Type predicates

- is.logical(x) returns TRUE if x has type logical. • is.character(x) - returns TRUE if x has type character.
- is.integer(x) returns TRUE if x has type integer.

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

- is.integer(1) is.double(1)
 - ## [1] TRUE
 - is.double(1L)
 - - ## [1] FALSE

is.double(3:8)

[1] FALSE

- ## [1] TRUE

is.numeric(1)

- is.numeric(1L)
- ## [1] TRUE
- is.numeric(3:7)

- ## [1] TRUE
- Mas/44 there is eful predicates: is.double(), is.atomic(), is.list(), is.vector(), and
- ## [1] TRUE

[1] FALSE

is.integer(1L)

is.integer(3:7)

[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 the rule that atomic vectors must always contain values of the same type.

```
c(1, "Hello")

## [1] "1"     "Hello"

c(FALSE, 3L)

## [1] 0 3

c(1.2, 3L)
```



[1] 1.2 3.0

Operator coercion

Operators and functions will generally attempt to coerce values to an appropriate type for the

given operation.	
3.1+1L	log(1)











FALSE | !5 ## [1] TRUE ## [1] FALSE Universiti Brunei Darrussalam

5 + FALSE

[1] 5

TRUE & 7

TRUE & FALSE

[1] FALSE

Explicit coercion

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

as.logical(5.2)	as.numeric(FALSE)
## [1] TRUE	## [1] 0
as.character(TRUE)	as.double("7.2")
## [1] "TRUE"	## [1] 7.2
as.integer(pi)	as.double("one")
## [1] 3	## [1] NA



Atomic Vectors

Conditionals & Control Flow

Error Checking

Missing Values

Functions

Loops

Logical (boolean) operators

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



Vectorized?

```
x <- c(TRUE, FALSE, TRUE)
y <- c(FALSE, TRUE, TRUE)
```

 $x \mid \mid y$

x && y

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

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

Note: both $| \ |$ and && only use the first value in the vector, all other values are ignored, there is no warning about the ignored values.

 $x \mid y$

x & y

Vectorization and math

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

$$c(1, 2, 3) + c(3, 2, 1)$$
 $log(c(1, 3, 0))$

$$c(1, 2, 3) / c(3, 2, 1)$$
 $sin(c(1, 2, 3))$

Length coercion (aka recycling)

```
x <- c(TRUE, FALSE, TRUE)
y <- TRUE
z <- c(FALSE, TRUE)
```

хІу

[1] TRUE TRUE TRUE

[1] TRUE FALSE TRUE

[1] TRUE TRUE TRUE

x & y

 $x \mid z$

15 / 44

y | z

y & z

[1] TRUE TRUE

[1] FALSE TRUE

Universiti Brunei Darrussalam

Length coercion and math

The same length coercion rules apply for most basic mathematical operators as well.

```
x \leftarrow c(1, 2, 3)
y < -c(5, 4)
z <- 10L
```

```
x + x
                                                  log(x)
```

```
## [1] 2 4 6
                                                ## [1] 0.0000000 0.6931472 1.0986123
```



[1] 1 2 3

x + z

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) ¹



¹Over 'x' here means the returned value will have the same length as 'x'.

Comparisons

```
x <- c("A","B","C")
z <- "A"

x == z

## [1] TRUE FALSE FALSE

## [1] TRUE FALSE FALSE

z %in% x</pre>
```

[1] TRUE

x > z

[1] FALSE TRUE TRUE

[1] FALSE TRUE TRUE

Conditional control flow

Conditional execution of code blocks is achieved via if statements.

```
x < -c(1, 3)
if (3 %in% x)
                                                 if (5 %in% x)
  print("Contains 3!")
                                                   print("Contains 5!")
## [1] "Contains 3!"
                                                 if (5 %in% x) {
                                                   print("Contains 5!")
if (1 %in% x)
                                                } else {
  print("Contains 1!")
                                                   print("Does not contain 5!")
   [1] "Contains 1!"
                                                 ## [1] "Does not contain 5!"
```

if is not vectorized

```
x < -c(1, 3)
if (x == 1)
  print("x is 1!")
## Error in if (x == 1) print("x is 1!"): the condition has length > 1
if (x == 3)
  print("x is 3!")
## Error in if (x == 3) print("x is 3!"): the condition has length > 1
```



Collapsing logical vectors

 $all(x \ge 2)$

21 / 44

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

```
any, all
x \leftarrow c(3,4,1)
```

x >= 2 $any(x \le 4)$

 $any(x \ge 2)$ ## [1] TRUE

```
if (any(x == 3))
                                                  print("x contains 3!")
## [1] FALSE
```

[1] "x contains 3!" Universiti Brunei

else if and else

```
x <- 3

if (x < 0) {
    "x is negative"
} else if (x > 0) {
    "x is positive"
} else {
    "x is zero"
}
x <- 0</pre>

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

[1] "x is positive"

22 / 44

[1] "x is zero"

if and return

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

```
x <- 5
                                                   x <- 5
s \leftarrow if (x \% 2 == 0) {
                                                   if (x \% 2 == 0) {
 x / 2
                                                     s <- x / 2
} else {
                                                   } else {
  3 * x + 1
                                                     s < -3 * x + 1
```

[1] 16

Notice that conditional expressions are evaluated in the parent scope.



[1] 16

Atomic Vectors

Conditionals & Control Flow

Error Checking

Missing Values

Functions

Loops

stop and stopifnot

Often we want to validate user input or function arguments - if our assumptions are not met then we often want to report the error and stop execution.

```
if (!ok)
stop("Things are not ok.")
```

Error in eval(expr, envir, enclos): Things are not ok.

```
stopifnot(ok)
```

ok <- FALSE

Error in eval(expr, envir, enclos): ok is not TRUE

Style choices

Do stuff:

```
if (condition_one) {
    ##
    ## Do stuff
    ##
} else if (condition_two) {
    ##
    ## Do other stuff
    ##
} else if (condition_error) {
    stop("Condition error occured")
}
```

Do stuff (better):

```
# Do stuff better
if (condition_error) {
  stop("Condition error occured")
}
if (condition_one) {
  ##
  ## Do stuff
  ##
} else if (condition_two) {
  ##
  ## Do other stuff
  ##
```

Atomic Vectors

Conditionals & Control Flow

Error Checking

Missing Values

Functions

Loops

Missing Values

typeof(NA + 1)

[1] "double"

26 / 44 " cter"

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.	
typeof(NA)	<pre>typeof(NA_character_)</pre>

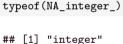
[1] "character"

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











Universiti Brunei

[1] "integer"

typeof(NA + 1L)

[1] "complex"

NA "stickiness"

NA

[1] NA

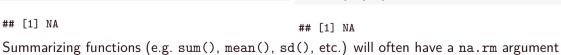
Because NAs represent missing values it makes sense that any calculation using them should also be missing.

```
1 + NA
                                                  sqrt(NA)
## [1] NA
                                                  ## [1] NA
```

	2-3
/ NA	3 ^ NA



[1] NA	## [1] NA
* 5	sum(c(1, 2, 3, NA))



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

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 that 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.

```
TRUE & NA
```

[1] NA FALSE & NA

TABBL & NA

[1] FALSE
TRUE | NA

[1] TRUE

FALSE | NA

[1] NA

Conditionals and missing values

NAs can be problematic in some cases (particularly for control flow)

```
1 == NA
## [1] NA
if (2 != NA)
  "Here"
## Error in if (2 != NA) "Here": missing value where TRUE/FALSE needed
if (all(c(1, 2, NA, 4) >= 1))
  "There"
## Error in if (all(c(1, 2, NA, 4) >= 1)) "There": missing value where TRUE/FALSE needed
if (any(c(1, 2, NA, 4) >= 1))
  "There"
```

[1] "There"

Testing for NA

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

NA == NA	is.na(c(1, 2, 3, NA))
## [1] NA	## [1] FALSE FALSE TRUE
is.na(NA)	any(is.na(c(1, 2, 3, NA)))
## [1] TRUE	## [1] TRUE
1 (4)	
is.na(1)	all(is.na(c(1, 2, 3, NA)))
HH [4] DATOR	
## [1] FALSE	## [1] FALSE

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

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

[1] NaN ## [1] Inf

[1] NaN ## [1] NaN

[1] Inf 31 / 44

0 / 0

[1] NaN

Testing for Inf and NaN

[1] FALSE

is.nan(-Inf)

[1] FALSE

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

, , , , , , , , , , , , , , , , , , ,	
<pre>is.finite(Inf)</pre>	is.finite(NaN)

[1] FALSE ## [1] FALSE

is.infinite(-Inf) is.infinite(NaN)

is.finite(NA)

[1] FALSE

is.nan(Inf) is.nan(NaN)

Coercion for infinity and NaN

First remember that Inf. -Inf. and NaN are doubles, however their coercion behavior is not

```
the same as for other doubles.
as.integer(Inf)
## [1] NA
```

as.integer(NaN)

[1] NA

[1] TRUE

as.logical(Inf)

as.logical(NaN)

[1] NA

[1] "Inf" as.character(NaN)

[1] "NaN"

as.character(Inf)

Atomic Vectors

Conditionals & Control Flow

Error Checking

Missing Values

Functions

Loops

Function parts

Functions are defined by two components: the arguments (formals) and the code (body). Functions are assigned names like any other object in R (using = or <-)

```
gcd \leftarrow function(x1, v1, x2 = 0, v2 = 0) {
  R <- 6371 # Farth mean radius in km
  acos(sin(y1) * sin(y2) + cos(y1) * cos(y2) * cos(x2 - x1)) * R # distance in km
typeof(gcd)
                           mode(gcd)
## [1] "closure"
                           ## [1] "function"
formals(gcd)
                            body(gcd)
## $x1
                            ## {
##
                                   R <- 6371
                            ##
                                   acos(sin(y1) * sin(y2) + cos(y1) * cos(y2) * cos(x2 - x1)) *
##
                            ##
## $y1
                                       R.
                            ##
                            ## }
                                                                                               Universiti
Brunei
Darvesslam
```

Return values

There are two approaches to returning values from functions in R.

Explicit: using one or more return function calls

```
f <- function(x) {
  return(x * x)
}
f(2)</pre>
```

Implicit: return value of the last expression is returned.

```
g <- function(x) {
   x * x
}
g(3)</pre>
```

[1] 9

[1] 4

Returning multiple values

If we want a function to return more than one value we can group things using atomic vectors or lists.

```
f <- function(x) c(x, x ^ 2, x ^ 3)
f(1:2)

## [1] 1 2 1 4 1 8

g <- function(x) list(x, "hello")
g(1:2)

## [[1]]</pre>
```

More on lists next time.

[1] 1 2

[[2]] ## [1] "hello"

36 / 44

##

Argument names

f(z = 1, x = 2, y = 3)

When defining a function we explicitly define names for the arguments, which become variables within the scope of the function. When calling a function we can use these names to pass arguments in an alternative order.

```
f <- function(x, y, z) {
  paste0("x = ", x, ", y = ", y, ", z = ", z)
}
f(1, 2, 3)

## [1] "x = 1, y = 2, z = 3"

## [1] "x = 1, y = 2, z = 3"

## [1] "x = 1, y = 2, z = 3"</pre>
```

f(v = 2, 1, x = 3)

```
## [1] "x = 2, y = 3, z = 1" ## [1] "x = 3, y = 2, z = 1" f(1, 2, 3, 4)
```

Error in f(1, 2, 3, 4): unused argument (4)



Argument defaults

It is also possible to give function arguments default values, so that they don't need to be provided every time the function is called.

```
f <- function(x, y = 1, z = 1) {
  paste0("x = ", x, ", y = ", y, ", z = ", z)
}
f(3)

f(z = 3, x = 2)

## [1] "x = 3, y = 1, z = 1"

## [1] "x = 2, y = 1, z = 3"</pre>
```

f(v = 2, 2)

```
## [1] "x = 3, y = 1, z = 1" ## [1] "x = 2, y = 2, z = 1"

f()
```

Error in paste0("x = ", x, ", y = ", y, ", z = ", z): argument "x" is missing, with no defar



f(x = 3)

Scope

R has generous scoping rules, if it can't find a variable in the current scope (e.g. a function's body) it will look for it in the next higher scope, and so on.

```
y <- 1
                                                       y <- 1
f <- function(x) {</pre>
                                                       g <- function(x) {</pre>
                                                          v <- 2
  x + y
                                                         x + y
f(3)
                                                       g(3)
## [1] 4
                                                       ## [1] 5
```

[1] 1

Universiti Brunei Darussalam

Scope (cont.)

Additionally, variables defined within a scope only persist for the duration of that scope, and do not overwrite variables at a higher scope.

```
x <- y <- z <- 1
f <- function() {
    y <- 2
    g <- function() {
      z < -3
      return(x + y + z)
    return(g())
f()
## [1] 6
```

```
c(x, y, z)
```

[1] 1 1 1 40 / 44 **Atomic Vectors**

Conditionals & Control Flow

Error Checking

Missing Values

Functions

Loops

for loops

Simplest, and most common type of loop in R-given a vector iterate through the elements and evaluate the code block for each.

```
is_even <- function(x) {
   res <- c()

   for(val in x) {
      res <- c(res, val %% 2 == 0)
   }

   res
}
is_even(1:10)</pre>
```

[1] FALSE TRUE FALSE TRUE FALSE TRUE FALSE TRUE

```
is_even(seq(1, 5, 2))
```

[1] FALSE FALSE FALSE

while loops

Repeat until the given condition is **not** met (i.e. evaluates to FALSE)

```
make_seq <- function(from = 1, to = 1, by = 1) {
  res <- c(from)
  cur <- from
  while(cur + by <= to) {</pre>
    cur = cur + by
    res = c(res, cur)
  res
make_seq(1, 6)
## [1] 1 2 3 4 5 6
```

... ...

make_seq(1, 6, 2)

[1] 1 3 5 42 / 44

Some helpful functions

Often we want to use a loop across the indexes of an object and not the elements themselves. There are several useful functions to help you do this: :, length, seq, seq along, seq len, etc.

```
4:7
                                                 seq_along(4:7)
## [1] 4 5 6 7
                                                 ## [1] 1 2 3 4
length(4:7)
                                                 seq_len(length(4:7))
## [1] 4
                                                 ## [1] 1 2 3 4
seq(4,7)
                                                 seq(4, 7, by = 2)
```

[1] 4 6



[1] 4 5 6 7

Avoid using 1:length(x)

A common loop construction you'll see in a lot of R code is using 1:length(x) to generate a

```
vector of index values for the vector x.
f <- function(x) {</pre>
                                                     g <- function(x) {</pre>
                                                       for(i in seq_along(x)) {
  for(i in 1:length(x)) {
                                                          print(i)
```

print(i) g(2:1)f(2:1)## [1] 1 ## [1] 1

```
## [1] 2
g(2)
```

[1] 1

[1] 2 f(2) ## [1] 1 f(integer()) g(integer()) 44 / 44

Universiti Brunei