SM-2302 Software for Mathematicians

R1: Logic and Types in R

Dr. Haziq Jamil
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).

R material lecture slides largely based off `https://sta323-sp22.github.io/`

Universiti Brunei Darussalam

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`)

```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
T
```

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

```r
T <- FALSE
T
```

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

character – text strings

Either single or double quotes are fine, 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"
```

Numeric types

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

```r
typeof(1.33)
```

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

```r
typeof(7)
```

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

```r
mode(1.33)
```

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

```r
mode(7)
```

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

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

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

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

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

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

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

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

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

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

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 *str*ucture of object `x`.

```r
typeof(1)
```

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

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

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

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

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

```r
typeof(TRUE)
```

```r
mode(1)
```

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

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

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

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

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

```r
mode(TRUE)
```

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

```r
str(1)
```

```
##  num 1
```

```r
str(1L)
```

```
##  int 1
```

```r
str("A")
```

```
##  chr "A"
```

```r
str(TRUE)
```

```
##  logi TRUE
```

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`.

```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
```

Many other useful predicates: `is.double()`, `is.atomic()`, `is.list()`, `is.vector()`, and

Universiti Brunei Darussalam

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.

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

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

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

```
## [1] 0 3
```

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

```
## [1] 1.2 3.0
```


ubd Universiti Brunei Darussalam

Operator coercion

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

```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")
```

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

Atomic Vectors

Conditionals & Control Flow

Error Checking

Missing Values

Functions

Loops

Logical (boolean) operators

<table>
<tr><th>Operator</th><th>Operation</th><th>Vectorized?</th></tr>
<tr><td>x | y</td><td>or</td><td>Yes</td></tr>
<tr><td><code>x & y</code></td><td>and</td><td>Yes</td></tr>
<tr><td><code>!x</code></td><td>not</td><td>Yes</td></tr>
<tr><td>x || y</td><td>or</td><td>No</td></tr>
<tr><td><code>x && y</code></td><td>and</td><td>No</td></tr>
<tr><td><code>xor(x, y)</code></td><td>exclusive or</td><td>Yes</td></tr>
</table>

Vectorized?

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

```
x | y
```

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

```
x || y
```

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

```
x & y
```

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

```
x && y
```

```
## [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.

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)
```

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

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

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

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

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

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

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

Length coercion (aka recycling)

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

```
x | y
```

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

```
y | z
```

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

```
x & y
```

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

```
y & z
```

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

```
x | z
```

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

Length coercion and math

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

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

```
x + x
```

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

```
x + z
```

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

```
log(x)
```

```
## [1] 0.0000000 0.6931472 1.0986123
```

```
y / z
```

```
## [1] 0.5 0.4
```

```
x %% y
```

```
## [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`)[1]

[1] Over ‘x‘ here means the returned value will have the same length as ‘x‘.

Comparisons

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

```r
x == z
```

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

```r
x != z
```

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

```r
x > z
```

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

```r
x %in% z
```

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

```r
z %in% x
```

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

Conditional control flow

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

```r
x <- c(1, 3)
```

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

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

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

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

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

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

```
## [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

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

```
x <- c(3,4,1)
```

```
x >= 2
```

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

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

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

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

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

```
x <
```

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

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

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

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

```
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 and return

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.

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.

```
ok <- FALSE
```

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

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

```
stopifnot(ok)
```

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

Style choices

Do stuff:

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

Do stuff (better):

```r
# 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

R uses `NA` to represent missing values in its data structures, what may not be obvious is that there are different `NA`s for 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"
```

NA "stickiness"

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

```r
1 + NA
```

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

```r
1 / NA
```

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

```r
NA * 5
```

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

```r
sqrt(NA)
```

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

```r
3 ^ NA
```

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

```r
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.

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

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 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
```

```
## [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)

```r
1 == NA
```

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

```r
if (2 != NA)
  "Here"
```

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

```r
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
```

```r
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
```

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

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

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

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

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

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

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

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

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

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

```
## [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

```r
pi / 0
```

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

```r
0 / 0
```

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

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

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

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

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

```r
NaN / NA
```

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

```r
NaN * NA
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
is.finite(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 for other doubles

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

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

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

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

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

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

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

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

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

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

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

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

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 <- function(x1, y1, x2 = 0, y2 = 0) {
  R <- 6371  # Earth mean radius in km
  acos(sin(y1) * sin(y2) + cos(y1) * cos(y2) * cos(x2 - x1)) * R   # distance in km
}
```

```
typeof(gcd)
```

```
## [1] "closure"
```

```
mode(gcd)
```

```
## [1] "function"
```

```
formals(gcd)
```

```
## $x1
##
##
## $y1
##
##
```

```
body(gcd)
```

```
## {
##     R <- 6371
##     acos(sin(y1) * sin(y2) + cos(y1) * cos(y2) * cos(x2 - x1)) *
##         R
## }
```

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)
```

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

Implicit: return value of the last expression is returned.

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

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

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

More on lists next time.

Argument names

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"
```

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

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

```
f(y = 2, 1, 3)
```

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

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

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

```
f(1, 2, 3, 4)
```

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


ubd Universiti Brunei Darussalam

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)
```

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

```
f(x = 3)
```

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

```
f(z = 3, x = 2)
```

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

```
f(y = 2, 2)
```

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

```
f()
```

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

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.

```r
y <- 1

f <- function(x) {
  x + y
}

f(3)
```

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

```r
y <- 1

g <- function(x) {
  y <- 2
  x + y
}

g(3)
```

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

```r
y
```

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

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
```

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)
```

```
##  [1] FALSE  TRUE 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) {
    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
```

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
```

```
## [1] 4 5 6 7
```

```
length(4:7)
```

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

```
seq(4,7)
```

```
## [1] 4 5 6 7
```

```
seq_along(4:7)
```

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

```
seq_len(length(4:7))
```

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

```
seq(4, 7, by = 2)
```

```
## [1] 4 6
```

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) {
  for(i in 1:length(x)) {
    print(i)
  }
}
f(2:1)
```

```
## [1] 1
## [1] 2
```

```
f(2)
```

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

```
f(integer())
```

```
g <- function(x) {
  for(i in seq_along(x)) {
    print(i)
  }
}
g(2:1)
```

```
## [1] 1
## [1] 2
```

```
g(2)
```

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

```
g(integer())
```