

# **Lec 02 - Logic and types in R**

## **Statistical Programming**

**Sta 323 | Spring 2022**

**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

<code>typeof()</code>	<code>mode()</code>
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. generic vectors have type `list`, but more on these later. See `?typeof` for more information.

# logical - boolean values (TRUE and FALSE)

```
typeof(TRUE)
```

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

```
typeof(FALSE)
```

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

```
mode(TRUE)
```

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

```
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.

```
T
```

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

```
T = FALSE
```

```
T
```

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

# character - **text strings**

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

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

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

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

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

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

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

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

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

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

```
"abc'123"
```

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

```
'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)

```
typeof(1.33)
```

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

```
typeof(7)
```

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

```
mode(1.33)
```

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

```
mode(7)
```

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

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

```
typeof( 7L )
```

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

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

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

```
mode( 7L )
```

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

```
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`.

```
typeof(1)
```

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

```
typeof(1L)
```

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

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

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

```
typeof(TRUE)
```

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

```
mode(1)
```

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

```
mode(1L)
```

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

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

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

```
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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
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 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
```

```
## [1] 4.1
```

```
5 + FALSE
```

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

```
TRUE & FALSE
```

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

```
TRUE & 7
```

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

```
log(1)
```

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

```
log(TRUE)
```

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

```
TRUE | FALSE
```

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

```
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.

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

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

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

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

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

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

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

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

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

```
## [1] 7.2
```

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

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

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

# 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.



# Conditionals & Control Flow

# Logical (boolean) operators

Operator	Operation	Vectorized?
<code>x   y</code>	or	Yes
<code>x &amp; y</code>	and	Yes
<code>!x</code>	not	Yes
<code>x    y</code>	or	No
<code>x &amp;&amp; y</code>	and	No
<code>xor(x, y)</code>	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
```

```
## [1] 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 = c(TRUE)
z = c(FALSE, TRUE)
```

```
x | y
```

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

```
x & y
```

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

```
y | z
```

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

```
y & z
```

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

```
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 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
```

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

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

# Comparison operators

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

over x here means the returned value will have the same length as x

# Comparisons

```
x = c("A", "B", "C")  
z = c("A")
```

```
x == z
```

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

```
x != z
```

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

```
x > z
```

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

```
x %in% z
```

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

```
z %in% x
```

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



# 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!")  
}
```

# if is not vectorized

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

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

```
## Warning in if (x == 1) print("x is 1!"): the condition has length > 1 and only  
## the first element will be used  
  
## [1] "x is 1!"
```

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

```
## Warning in if (x == 3) print("x is 3!"): the condition has length > 1 and only  
## the first element will be used
```

# 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 <= 4
```

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

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

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

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

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

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

## else if **and** else

```
x = 3

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

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

```
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.

```
x = 5
```

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

```
s
```

```
## [1] 16
```

```
x = 5
```

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

```
s
```

```
## [1] 16
```

Notice that conditional expressions are evaluated in the parent scope.

# Exercise 2

Take a look at the following code, without running it in 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")  
  }  
}
```

what do you expect the outcome will be for each of the following

```
f(1)  
f(3)  
f(8)  
f(-1)  
f(-3)  
f(1.2)
```

More on functions next time

# Error Checking

# 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: ok is not TRUE
```

Note - an error (like the one generated by `stop`) will prevent an RMarkdown document from compiling unless `error = TRUE` is set for that code chunk



# Style choices

Do stuff:

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

Do stuff (better):

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

# Exercise 3

Write a set of conditional(s) that satisfies the following requirements,

- If  $x$  is greater than 3 and  $y$  is less than or equal to 3 then print "Hello world!"
- Otherwise if  $x$  is greater than 3 print "!dlrow olleH"
- If  $x$  is less than or equal to 3 then print "Something else ..."
- Stop execution if  $x$  is odd and  $y$  is even and report an error, don't print any of the text strings above.

Test out your code by trying various values of  $x$  and  $y$ .