In R (almost)
everything is a vector

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 more detail later.

Vector types

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

```r
typeof(TRUE)
```

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

```r
mode(TRUE)
```

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

`character` - text strings

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

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

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

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

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

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

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

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

`double` - floating point numerical values (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 numerical values (indicated with an `L`)

```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 constructed 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 *always* 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`.

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

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

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

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

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

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

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

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

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

```r
is.list(list(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 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
```

Operator coercion

Operators and functions will also 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
```

Conditionals

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?

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

```r
x | y
```

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

```r
x || y
```

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

```r
x & y
```

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

```r
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.

```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

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

```r
x | y
```

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

```r
x & y
```

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

```r
y | z
```

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

Comparisons

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`)

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("This!")
```

```
## [1] "This!"
```

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

```
## [1] "That!"
```

```
if (5 %in% x)
  print("Other!")
```

`if` is not vectorized

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

```r
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!"
```

```r
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!")
```

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

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.

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

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

```r
stopifnot(ok)
```

```
## Error: ok is not TRUE
```

```r
stopifnot(is.logical(ok))
```

```r
stopifnot(is.logical(ok+0))
```

```
## Error: is.logical(ok + 0) is not TRUE
```

Style choices

Simple is usually better than complicated - generally it is better to have fewer clauses and have the more important conditions first (e.g. failure conditions)

Do stuff (ok):

```
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
  ##
}
```

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 the different types.

```r
typeof(NA)
```

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

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

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

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

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

```r
typeof(NA_character_)
```

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

```r
typeof(NA_real_)
```

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

```r
typeof(NA_integer_)
```

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

NA contageon

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

```
1 + NA
```

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

```
1 / NA
```

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

```
NA * 5
```

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

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

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

```
sqrt(NA)
```

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

```
3^NA
```

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

NAs are not always contageous

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.

```r
TRUE & NA
```

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

```r
FALSE & NA
```

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

```r
TRUE | NA
```

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

```r
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`).

```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
```

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

```r
NA
```

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

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

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

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

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

```r
1/0 == Inf
```

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

```r
-1/0 == Inf
```

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

```r
is.finite(1/0+1/0)
```

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

```r
is.finite(1/0-1/0)
```

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

```r
is.nan(1/0-1/0)
```

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

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

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

```r
is.nan(NA)
```

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

Coercion for infinity and NaN

First remember that `Inf`, `-Inf`, and `NaN` have type double, however their coercion behavior is not the same as for 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(NaN)
```

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

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

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

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

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

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.

```
res = c()
for(x in 1:10) {
  res = c(res, x^2)
}
res
```

```
##  [1]   1   4   9  16  25  36  49  64  81 100
```

```
res = c()
for(y in list(1:3, LETTERS[1:7], c(TRUE,FALSE))) {
  res = c(res, length(y))
}
res
```

```
## [1] 3 7 2
```

`while` loops

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

```r
i = 1
res = rep(NA,10)

while (i <= 10) {
  res[i] = i^2
  i = i+1
}

res
```

```
##  [1]   1   4   9  16  25  36  49  64  81 100
```

`repeat` loops

Repeat the loop until a `break` is encountered

```r
i = 1
res = rep(NA,10)

repeat {
  res[i] = i^2
  i = i+1
  if (i > 10)
    break
}

res
```

```
##  [1]   1   4   9  16  25  36  49  64  81 100
```

Special keywords - `break` and `next`

These are special actions that only work *inside* of a loop

- `break` - ends the current **loop** (inner-most)
- `next` - ends the current **iteration**

```r
res = c()
for(i in 1:10) {
    if (i %% 2 == 0)
        break
    res = c(res, i)
    print(res)
}
```

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

```r
res = c()
for(i in 1:10) {
    if (i %% 2 == 0)
        next
    res = c(res,i)
    print(res)
}
```

```
## [1] 1
## [1] 1 3
## [1] 1 3 5
## [1] 1 3 5 7
## [1] 1 3 5 7 9
```

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.

```r
4:7
```

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

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

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

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

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

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

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

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

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

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

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

Acknowledgments

Above materials are derived in part from the following sources:

- Hadley Wickham - Advanced R
- R Language Definition