Lecture 01

Colin Rundel

2019-09-16

Course Details

Course website

Learn - https://learn.ed.ac.uk

and/or

https://statprog-s1-2019.github.io

Reproducible Research / Computing

- R + RStudio + rmarkdown
- Git + github

Reproducible Research / Computing

- R + RStudio + rmarkdown
- Git + github

Programming course with statistics

vs.

Statistics course with programming

Weekly Schedule

- Mondays, 16:10 - 18:00
    - Lecture
- Thursday, ??? - ???
    - Workshop

Marking

Assignment	Type	Value	Assigned
Homework 1	Team	10%	Out Week 2
Homework 2	Team	10%	Out Week 4
Project 1	Individual	30%	Out Week 5
Homework 3	Team	10%	Out Week 7
Homework 4	Team	10%	Out Week 9
Project 2	Individual	30%	Out Week 10

Teams

- Team homework assignments
  - Roughly biweekly assignments
  - Open ended
  - 5 - 20 hours of work
  - Peer evaluation at the end

- Expectations and roles
  - Everyone is expected to contribute equal *effort*
  - Everyone is expected to understand *all* code turned in
  - Individual contribution evaluated by peer evaluation, commits, etc.

Collaboration policy

- Only work that is clearly assigned as team work should be completed collaboratively (Homework).
- On projects you may not directly share or discuss code with anyone other than the Instructors and Tutors
- On homeworks you may not directly share code with other team(s) in this class, however you are welcome to discuss the problems together and ask for advice

Sharing / reusing code policy

- I am well aware that a huge volume of code is available on the web to solve any number of problems.
- Unless I explicitly tell you not to use something the course's policy is that you may make use of any online resources (e.g. Google, StackOverflow, etc.) but you must explicitly cite where you obtained any code you directly use (or use as inspiration).
- Any recycled code that is discovered and is not explicitly cited will be treated as plagiarism.

(Almost)
Everything is a Vector

Vectors

The fundamental building block of data in R are vectors (collections of related values, objects, other data structures, etc).

Vectors

The fundamental building block of data in R are vectors (collections of related values, objects, other data structures, etc).

R has two types of vectors:

- **atomic** vectors
    - homogeneous collections of the *same* type (e.g. all logical values, all numbers, or all character strings).
- **generic** vectors
    - 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:

`typeof`	`mode`
logical	logical
double	numeric
integer	numeric
character	character
complex	complex
raw	raw

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.

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

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

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!"
```

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

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

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

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

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

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

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

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

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

Type Coercion

R is a dynamically typed language -- it will automatically convert between most type without raising warnings or errors.

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

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

Type Coercion

R is a dynamically typed language -- it will automatically convert between most type without raising warnings or errors.

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

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

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

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

Type Coercion

R is a dynamically typed language -- it will automatically convert between most type without raising warnings or errors.

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

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

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

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

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

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

Operator coercion

Functions and operators will attempt to coerce object to an appropriate type

```r
3.1+1L
```

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

Operator coercion

Functions and operators will attempt to coerce object to an appropriate type

```
3.1+1L
```

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

```
log(TRUE)
```

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

Operator coercion

Functions and operators will attempt to coerce object to an appropriate type

```
3.1+1L
```

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

```
log(TRUE)
```

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

```
TRUE & 7
```

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

Operator coercion

Functions and operators will attempt to coerce object to an appropriate type

```
3.1+1L
```

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

```
log(TRUE)
```

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

```
TRUE & 7
```

```
## [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
```

Conditionals

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>x & y</td><td>and</td><td>Yes</td></tr>
<tr><td>!x</td><td>not</td><td>Yes</td></tr>
<tr><td>x || y</td><td>or</td><td>No</td></tr>
<tr><td>x && y</td><td>and</td><td>No</td></tr>
<tr><td>xor(x,y)</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
```

Vectorization and arithmatic

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

```
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

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

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
```

Length coercion

```
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
```

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
```

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)
```

Conditional Control Flow

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

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

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

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

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!"
```

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!")
```

Note `if` is not vectorized

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

Note `if` is not vectorized

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

```
if (x %in% 3)
  print("Now Here!")
```

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

Note `if` is not vectorized

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

```
if (x %in% 3)
  print("Now Here!")
```

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

```
if (x %in% 1)
  print("Now Here!")
```

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

```
## [1] "Now Here!"
```

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
```

Nesting Conditionals

```r
x = 3
if (x < 0) {
  "Negative"
} else if (x > 0) {
  "Positive"
} else {
  "Zero"
}
```

```
## [1] "Positive"
```

```r
x = 0
if (x < 0) {
  "Negative"
} else if (x > 0) {
  "Positive"
} else {
  "Zero"
}
```

```
## [1] "Zero"
```

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 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 `NA`s 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"
```

Stickiness of Missing Values

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

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

```r
sqrt(NA)
```

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

```r
3^NA
```

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

Conditionals and missing values

`NA`s can be problematic in some cases (particularly for control flow)

```
1 == NA
```

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

Conditionals and missing values

`NA`s 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
```

Conditionals and missing values

`NA`s 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
```

Conditionals and missing values

`NA`s 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`).

```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 (double) values

- `NaN` - Not a number
- `Inf` - Positive infinity
- `-Inf` - Negative infinity

```
pi / 0
```

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

```
0 / 0
```

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

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

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

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

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

```
NaN / NA
```

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

```
NaN * NA
```

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

Testing for `inf` and `NaN`

`NaN` and `Inf` don't have the same testing issues that `NA` has, but there are still convenience functions for testing for

```r
NA
```

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

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

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

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

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

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

```
## [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
```

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 double values.

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

```
## Warning: NAs introduced by coercion to integer range
```

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

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

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

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

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

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

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

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

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

```r
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.

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
```

`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
```

Note - the code above is terrible for several reasons, you should never write anything that looks like this

`while` loops

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

```
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 until `break`

```
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 helper 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
```

Exercise 2

Below is a vector containing all prime numbers between 2 and 100:

```
primes = c( 2,  3,  5,  7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
           43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97)
```

If you were given the vector `x = c(3,4,12,19,23,51,61,63,78)`, write the R code necessary to print only the values of `x` that are *not* prime (without using subsetting or the `%in%` operator).

Your code should use *nested* loops to iterate through the vector of primes and `x`.

Acknowledgments

Above materials are derived in part from the following sources:

- Hadley Wickham - Advanced R
- R Language Definition