Vectorization & Control Flow

Lecture 02

Dr. Colin Rundel

From last time - Exercise 1

Part 1

What is the type of the following vectors? Explain why they have that type.

```
1 c(1, NA+1L, "C")
2 c(1L / 0, NA)
3 c(1:3, 5)
4 c(3L, NaN+1L)
5 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)?

Logical & Comparison operators

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?

```
1 x = c(TRUE,FALSE,TRUE)
2 y = c(FALSE,TRUE,TRUE)
```

```
1 x | y
```

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

```
1 x & y
```

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

```
1 x || y
```

```
Error in `x || y`:
! 'length = 3' in coercion to 'logical(1)'
```

```
1 x && y
```

```
Error in `x && y`:
! 'length = 3' in coercion to 'logical(1)'
```

```
1 TRUE && FALSE
```

```
[1] FALSE
```

Before R 4.3 both `||` and `&&` only used the *first* value in the vector, all other values are ignored and there was no warning about the ignored values.

`&` and `|` are almost always going to be the right choice, the only time we use `&&` or `||` is when you need to take advantage of short-circuit evaluation.

Vectorization and math

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

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

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

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

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

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

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

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

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

Length coercion (aka recycling)

If the lengths of the vector do not match, then the shorter vector has its values recycled to match the length of the longer vector.

```
1 x = c(TRUE, FALSE, TRUE)
2 y = c(TRUE)
3 z = c(FALSE, TRUE)
```

```
1 x | y
```

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

```
1 y | z
```

```
[1] TRUE TRUE
```

```
1 x & y
```

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

```
1 y & z
```

```
[1] FALSE  TRUE
```

```
1 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,

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

```
x + x
```

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

```
y / z
```

```
[1] 0.5 0.4
```

```
x + z
```

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

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

```
[1] 10.00000 10.69315 11.09861
```

```
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?
`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

```
1 x = c("A","B","C")
2 y = c("A")
```

```
1 x == y
```

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

```
1 x != y
```

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

```
1 x %in% y
```

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

```
1 y %in% x
```

```
[1] TRUE
```

Type coercion also applies for comparison operators which can result in *interesting* behavior

```
1 TRUE == "TRUE"
```

```
[1] TRUE
```

```
1 FALSE == 1
```

```
[1] FALSE
```

```
1 TRUE == 1
```

```
[1] TRUE
```

```
1 TRUE == 5
```

```
[1] FALSE
```

> & < with characters

While perhaps unexpected, these comparison operators can be used with character values.

```
"A" < "B"
```

```
[1] TRUE
```

```
"A" > "B"
```

```
[1] FALSE
```

```
"A" < "a"
```

```
[1] FALSE
```

```
"a" > "!"
```

```
[1] TRUE
```

```
"Good" < "Goodbye"
```

```
[1] TRUE
```

```
c("Alice", "Bob", "Carol") <
```

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

Note - to better understand how this works, i.e. the ordering used, see ASCII code and

Control Flow

Conditional Control Flow

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

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

```
1 if (3 %in% x) {
2   print("Contains 3!")
3 }
```

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

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

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

```
1 if (5 %in% x) {
2   print("Contains 5!")
3 }
```

```
1 if (5 %in% x) {
2   print("Contains 5!")
3 } else {
4   print("Does not contain 5!
5 }
```

```
[1] "Does not contain 5!"
```

`if` is *not* vectorized

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

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

```
Error in `if (x == 1) ...`:
! the condition has length > 1
```

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

```
Error in `if (x == 3) ...`:
! the condition has length > 1
```

This behavior (throwing an error) was added in R 4.2, previous versions will only emit a warnings (while using the first value in the condition vector).

Collapsing logical vectors

There are a couple of helpful functions for collapsing logical vectors: `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!"
```

`else if` and `else`

```
1 x = 3
2
3 if (x < 0) {
4   "x is negative"
5 } else if (x > 0) {
6   "x is positive"
7 } else {
8   "x is zero"
9 }
```

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

```
1 x = 0
2
3 if (x < 0) {
4   "x is negative"
5 } else if (x > 0) {
6   "x is positive"
7 } else {
8   "x is zero"
9 }
```

```
[1] "x is zero"
```

`if` blocks return a value

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
```

Exercise 1

Take a look at the following code below on the left, without running it in R what do you expect the outcome will be for each call on the right?

```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")
  }
}
```

```r
f(1)
f(3)
f(-1)
f(-3)
f(1:2)
f("0")
f("zero")
```

Conditionals and missing values

NAs can be particularly problematic for control flow,

```
1 if (2 != NA) {
2   "Here"
3 }
```

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

```
1 2 != NA
```

```
[1] NA
```

```
1 if (all(c(1,2,NA,4) >= 1)) {
2   "There"
3 }
```

```
Error in `if (all(c(1, 2, NA, 4) >= 1)) ..
! missing value where TRUE/FALSE needed
```

```
1 all(c(1,2,NA,4) >= 1)
```

```
[1] NA
```

```
1 if (any(c(1,2,NA,4) >= 1)) {
2   "There"
3 }
```

```
[1] "There"
```

```
1 any(c(1,2,NA,4) >= 1)
```

```
[1] TRUE
```

Testing for `NA`

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

```
1 NA == NA
```

```
[1] NA
```

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

```
[1] TRUE
```

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

```
[1] FALSE
```

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

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

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

```
[1] TRUE
```

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

```
[1] FALSE
```

Note `is.na()` is testing for a property of the *values*, **not** a property of the *vector* - so it is *vectorized*.

Errors

`stop` and `stopifnot`

Often we want to validate user input, function arguments, or other assumptions in our code - if our assumptions are not met then we often want to report (throw) an error and stop execution.

```
ok = FALSE
```

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

```
Error:
! Things are not ok.
```

```
stopifnot(ok)
```

```
Error:
! ok is not TRUE
```

```
stopifnot("Still not ok" = ok)
```

```
Error:
! Still not ok
```

Note - an error (like the one generated by `stop`) will prevent an RMarkdown or Quarto document from rendering unless

Style choices

Do stuff:

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

Do stuff (better):

```
# Do stuff better
if (condition_error) {
  stop("Condition error occurre
}

if (condition_one) {
  ## Do stuff
} else if (condition_two) {
  ## Do other stuff
}
```

Why errors?

R has a variety of different output "methods" that can be used,

- Printed output - `cat()`, `print()`
- Diagnostic messages - `message()`
- Warnings - `warning()`
- Errors - `stop()`, `stopifnot()`

Each of these provides text output while also providing signals which can be interacted with programmatically (e.g. catching errors or treating warnings as errors).

Handling errors

```
flip = function() {
  if (runif(1) > 0.5)
    stop("Heads")
  else
    "Tails"
}
```

```
flip()
```

```
[1] "Tails"
```

```
flip()
```

```
Error in `flip()`:
! Heads
```

```
x = try(flip(), silent=TRUE)
str(x)
```

```
 chr "Tails"
```

```
x = try(flip(), silent=TRUE)
str(x)
```

```
 'try-error' chr "Error in flip() : Heads\
 - attr(*, "condition")=List of 2
  ..$ message: chr "Heads"
  ..$ call   : language flip()
  ..- attr(*, "class")= chr [1:3] "simpleE
```

Functions

What is a function?

Functions are abstractions in programming languages that allow us to modularize our code into small “self contained” units.

In general the goals of writing functions is to,

- Simplify a complex process or task into smaller sub-steps
- Allow for the reuse of code without duplication
- Improve the readability of your code
- Improve the maintainability of your code

Functions as objects

Functions are first-class objects in R and have a *mode* of `function`. They are assigned names like other objects using `=` or `<-`.

```
gcd = function(x1, y1, x2 = 0, y2 = 0) {
  # x and y values should be in radians
  R = 6371 # Earth mean radius in km

  # distance in km
  acos(sin(y1)*sin(y2) + cos(y1)*cos(y2) * cos(x2-x1)) * R
}
```

```
typeof(gcd)
```

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

```
mode(gcd)
```

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

Function elements

In R functions are defined by *two* components:

1. the arguments (`formals`)
2. the code / expression (`body`).

```
1 str( formals(gcd) )
```

```
Dotted pair list of 4
 $ x1: symbol
 $ y1: symbol
 $ x2: num 0
 $ y2: num 0
```

```
1 body(gcd)
```

```
{
    R = 6371
    acos(sin(y1) * sin(y2) + cos(y1
        R
}
```

```
1 typeof( formals(gcd) )
```

```
[1] "pairlist"
```

```
1 typeof( body(gcd) )
```

```
[1] "language"
```

Note when using `body()` here the code we get back has had comments removed, if you want to access the full code you

Return values

As with most other languages, functions are most often used to process inputs and return a value as output. There are two approaches to returning values from functions in R - *explicit* and *implicit* returns.

Explicit - using one or more `return` 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
```

Invisible returns

Many functions in R make use of an invisible return value

```
1 invisible(1)
```

```
1 print(invisible(1))
```

```
[1] 1
```

```
1 f = function(x) {
2   x
3 }
```

```
1 g = function(x) {
2   invisible(x)
3 }
```

```
1 f(1)
```

```
[1] 1
```

```
1 g(1)
```

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

```
[1] 1
```

```
1 z = g(1)
2 z
```

```
[1] 1
```

Returning multiple values

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

```r
f = function(x) {
  c(x, x^2, x^3)
}

f(1:2)
```

```
[1] 1 2 1 4 1 8
```

```r
g = function(x) {
  list(x, "hello")
}

g(1:2)
```

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

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

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

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

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

```
Error in `f()`:
! unused argument (4)
```

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

```
Error in `f()`:
! unused argument (m = 3)
```

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.

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

```
1 f(3)
```

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

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

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

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

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

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

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

```
1 f()
```

```
Error in `f()`:
! argument "x" is missing, with no default
```

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 until an object with that name is found or it runs out of environments.

```
1 y = 1
2
3 f = function(x) {
4   x + y
5 }
```

```
1 f(3)
```

```
[1] 4
```

```
1 y = 1
2
3 g = function(x) {
4   y = 2
5   x + y
6 }
```

```
1 g(3)
```

```
[1] 5
```

```
1 y
```

```
[1] 1
```

Scope persistance

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

```
x = 1
y = 1
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
```

Lazy evaluation

Another interesting / unique feature of R is that function arguments are lazily evaluated, which means they are only evaluated when needed.

```
1 f = function(x) {
2   TRUE
3 }
```

```
1 f(1)
```

```
[1] TRUE
```

```
1 f(stop("Error"))
```

```
[1] TRUE
```

```
1 g = function(x) {
2   x
3   TRUE
4 }
```

```
1 g(1)
```

```
[1] TRUE
```

```
1 g(stop("Error"))
```

```
Error in `g()`:
! Error
```

More “practical” lazy evaluation

The previous example is not particularly useful, a more common use for this lazy evaluation is that this enables us define arguments as expressions of other arguments.

```
f = function(x, y=x+1, z=1) {
  x = x + z
  y
}
```

```
f(x=1)
```

```
[1] 3
```

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

```
[1] 4
```

Operators as functions

In R, operators are actually a special type of function - using backticks around the operator we can write them as functions.

```
1 `+`
```

```
function (e1, e2)  .Primitive("+")
```

```
1 typeof(`+`)
```

```
[1] "builtin"
```

```
1 x = 4:1
2 x + 2
```

```
[1] 6 5 4 3
```

```
1 `+`(x, 2)
```

```
[1] 6 5 4 3
```

Getting Help

Prefixing any function name with a `?` will open the related help file for that function.

```
?`+`
?sum
```

For functions not in the base package, you can generally see their implementation by entering the function name without parentheses (or using the `body` function).

```
lm
```

```
function (formula, data, subset, weights, na.action, method = "qr",
    model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE,
    contrasts = NULL, offset, ...)
{
    ret.x <- x
    ret.y <- y
    cl <- match.call()
    mf <- match.call(expand.dots = FALSE)
    m <- match(c("formula", "data", "subset", "weights", "na.action",
        "offset"), names(mf), 0L)
    mf <- mf[c(1L, m)]
    mf$drop.unused.levels <- TRUE
    mf[[1L]] <- quote(stats::model.frame)
    mf <- eval(mf, parent.frame())
```

```
if (method == "model.frame")
    return(mf)
else if (method != "qr")
    warning(gettextf("method = '%s' is not supported. Using 'qr'",
        method), domain = NA)
mt <- attr(mf, "terms")
```

Less Helpful Examples

```
1 list
```

```
function (...)  .Primitive("list")
```

```
1 `[`
```

```
.Primitive("[")
```

```
1 sum
```

```
function (..., na.rm = FALSE)  .Primitive("sum")
```

```
1 `+`
```

```
function (e1, e2)  .Primitive("+")
```

Infix functions (operators)

We can define our own infix functions like `+` or `*`, the only requirement is that the function name must start and end with a `%`.

```
1 `%nand%` = function(x, y) {
2   !(x & y)
3 }
```

```
1 TRUE %nand% TRUE
```

```
[1] FALSE
```

```
1 FALSE %nand% TRUE
```

```
[1] TRUE
```

```
1 TRUE %nand% FALSE
```

```
[1] TRUE
```

```
1 FALSE %nand% FALSE
```

```
[1] TRUE
```

Replacement functions

We can also define functions that allow for 'inplace' modification like `attr` or `names`.

```
`last<-` = function(x, value) {
  x[length(x)] = value
  x
}
```

```
x = 1:10
```

```
last(x) = 5L
x
```

```
[1] 1 2 3 4 5 6 7 8 9 5
```

```
last(x) = NA
x
```

```
[1]  1  2  3  4  5  6  7  8  9 NA
```

```
last(1)
```

```
Error in `last()`:
! could not find function "last"
```

```
`modify<-` = function(x, pos, value) {
  x[pos] = value
  x
}
```

```
x = 1:10
```

```
modify(x,1) = 5L
x
```

```
[1]  5  2  3  4  5  6  7  8  9 10
```

```
modify(x, 9:10) = 1L
x
```

```
[1] 5 2 3 4 5 6 7 8 1 1
```

Loops

for loops

These are the most common type of loop in R - given a vector it iterates through the elements and evaluate the code expression for each value.

```
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

This loop repeats evaluation of the code expression until the 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
```

`repeat` loops

Equivalent to a `while(TRUE){}` loop, it repeats until a `break` statement

```
make_seq2 = function(from = 1, to = 1, by = 1) {
  res = c(from)
  cur = from

  repeat {
    cur = cur + by
    if (cur > to)
      break
    res = c(res, cur)
  }

  res
}
```

```
make_seq2(1, 6)
```

```
[1] 1 2 3 4 5 6
```

```
make_seq2(1, 6, 2)
```

```
[1] 1 3 5
```

Special keywords - `break` and `next`

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

- `break` - ends the current **loop**

```
f = function(x) {
  res = c()
  for(i in x) {
    if (i %% 2 == 0)
      break
    res = c(res, i)
  }
  res
}
```

```
f(1:10)
```

```
[1] 1
```

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

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

- `next` - ends the current **iteration**

```
g = function(x) {
  res = c()
  for(i in x) {
    if (i %% 2 == 0)
      next
    res = c(res,i)
  }
  res
}
```

```
g(1:10)
```

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

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

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

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.

```
1 4:7
```

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

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

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

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

```
[1] 4
```

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

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

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

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

```
1 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`.

```
1 f = function(x) {
2   for(i in 1:length(x)) {
3     print(i)
4   }
5 }
```

```
1 f(2:1)
```

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

```
1 f(2)
```

```
[1] 1
```

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

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

```
1 g = function(x) {
2   for(i in seq_along(x)) {
3     print(i)
4   }
5 }
```

```
1 g(2:1)
```

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

```
1 g(2)
```

```
[1] 1
```

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

What was the problem?

```
1 length(integer())
```

```
[1] 0
```

```
1 1:length(integer())
```

```
[1] 1 0
```

```
1 seq_along(integer())
```

```
integer(0)
```

Exercise 2

To the right is a vector containing all prime numbers between 2 and 100 and a separate vector `x` containing some values we would like to check for primality.

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 will need to use *nested* loops to iterate through the vector of `primes` and `x`.

```
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)

x = c(3,4,12,19,23,51,61,63,78)
```