Lec 04 - NAs, functions, and loops

Statistical Programming

Sta 323 | Spring 2022

Dr. Colin Rundel

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 different atomic types.

```
typeof(NA)
```

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

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

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

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

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

```
typeof(c(NA,""))
```

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

```
typeof(NA_character_)
```

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

```
typeof(NA_real_)
```

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

```
typeof(NA_integer_)
```

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

```
typeof(NA_complex_)
```

```
## [1] "complex"
```

NA "stickiness"

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
```

```
sqrt(NA)
```

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

```
3^NA
```

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

```
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.

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

NAs are not always sticky

A useful mental model for NAs 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)

```
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`).

```
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

```
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`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
```

```
Inf > 1
```

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

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

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

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

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

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

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

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

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

```
is.infinite(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)
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Functions

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

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

```
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
##         R
## }
```

Return values

There are two approaches to returning values from functions in R - explicit and implicit returns.

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
```

Invisible returns

Many functions in R make use of an invisible return value

```
f = function(x) {
  print(x)
}

y = f(1)
```

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

```
y
```

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

```
g = function(x) {
  invisible(x)
}
```

```
g(2)
```

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

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]]
```

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(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(1, 2, 3, 4): unused argument (4)
```

```
f(1, 2, 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.

```
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 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.

```
y = 1

f = function(x) {
  x + y
}

f(3)
```

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

```
y = 1

g = function(x) {
  y = 2
  x + y
}

g(3)
```

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

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

```
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
```

Exercise 1 - scope

What is the output of the following code? Explain why.

```
z = 1

f = function(x, y, z) {
  z = x+y

  g = function(m = x, n = y) {
    m/z + n/z
  }

  z * g()
}

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

Lazy evaluation

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

```
f = function(x) {
  TRUE
}
```

```
f(1)
```

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

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

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

```
g = function(x) {
  x
  TRUE
}
```

```
g(1)
```

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

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

```
## Error in g(stop("Error")): Error
```

Operators as functions

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

```
`+`
```

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

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

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

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

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

```
`+`(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
```

Less Helpful Examples

```
list
```

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

```
`[`
```

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

```
sum
```

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

```
`+`
```

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

For the curious the lookup package will help you track down the source for these 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
```

`repeat` **loops**

Repeat the loop until a `break` is encountered

```
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)
```

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**

```
f = function(x) {
  res = c()

  for(i in x) {
    if (i %% 2 == 0)
      break

    res = c(res, i)
  }

  res
}

f(1:10)
```

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

```
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
```

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())
```

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

```
g = function(x) {
  for(i in seq_along(x)) {
    print(i)
  }
}

g(2:1)
```

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

```
g(2)
```

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

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

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`.