SCR Week 4: live coding

More about Functions

Scoping | Session 1

```
numerator_value <- 12

GiveRemainder <- function(divisor) {
   return(numerator_value %% divisor)
}
GiveRemainder(10)
numerator_value

# numerator_value <- 12
GiveRemainder(10)
```

Setting a value | Session 2

This is part of an Object Oriented Programming style

```
person <- list(name = "Who?", age = 34, gender = "male")
person[["name"]]
```

```
## [1] "Who?"
```

```
GetName <- function(x) {
  return(x[['name']])
}
GetName(person)
```

```
## [1] "Who?"
```

```
GetName(person) <- ""
```

```
## Error in GetName(person) <- "": could not find function "GetName<-"
```

However, apparantly in R, we can assign values based on a function call.These are functions as `names()`, `class()`, `mode()`...

```
obj <- c(a = 2, b  = 3, c = 3)
names(obj)
```

```
## [1] "a" "b" "c"
```

```
names(obj) <- paste("V", 1:3, sep = "")
names(obj)
```

```
## [1] "V1" "V2" "V3"
```

The trick on how to do this... (will be skipped if not enough time!)

```
MyName <- 'MyName<-' <- function(x, value = x[['name']]){
  x[['name']] <- value
  return(x)
}
```

```
MyName(person) <- ""
MyName(person)
```

```
## $name
## [1] ""
##
## $age
## [1] 34
##
## $gender
## [1] "male"
```

Parts of a function | Session 3

```
Multiply <- function(){}
```

```
args(Multiply)
```

```
## function ()
## NULL
```

```
formals(Multiply)
```

```
## NULL
```

```
Multiply <- function(x, y = x, ...){

  args <- list(...) # remember swirl?

  if (!is.null(args$message)){ # remember tries to find a named entry / argument?
    print(args$message)
  }

  return_value <- x * y
  return(return_value)
}
```

```r
args(Multiply)
```

```
## function (x, y = x, ...)
## NULL
```

```r
formals(Multiply)
```

```
## $x
##
##
## $y
## x
##
## $...
```

```r
Multiply(2)
```

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

```r
Multiply(2, 4)
```

```
## [1] 8
```

```r
Multiply(x = 2, y = 4, message = "This is a multiplication of 2 and 4")
```

```
## [1] "This is a multiplication of 2 and 4"
```

```
## [1] 8
```

Down the rabbit hole

When getting help on a function, don't forget to look at the examples at the end of the file.

```r
?mean
example(mean) # or just call them.
```

Other options:

```r
View(Multiply)
Multiply <- edit(Multiply)
Multiply(2,3, message ="hi")
```

R is open source!

```r
View(apply) # oh dear
```

Something simpler then?

```r
View(mean)
methods("mean")
View(mean.default)
?.Internal
.Internal(mean(1:100))
View(.Internal)
.Primitive(".Internal")
```

To really find out how the function `mean()` works, download the source of R (e.g. not the binary versions…). Then, find the file: names.c, and look for "mean", and find "do_summary". So, we have to go to the `summary.c` file and look for do_summary…? pff

Can we find out the source code for `rnorm()`…?

```r
View(rnorm) # this is new ?
```

`.Call?` Darn it! We have to go -> names.c -> etc..

Whenever you encounter `C()`, `.Call()`, `.Fortran()`, `.External()`, or `.Internal()` and `.Primitive()`, don't bother (for this course!)

Factors | Session 4

```r
set.seed(20180928)
my_vector <- letters[sample(1:10, 25, replace=T)]
# class(my_vector)

# creating a factor
# my_factor <- factor(my_vector)
my_factor <- factor(my_vector, levels = letters[1:10])
attributes(my_factor)
levels(my_factor) # works like a names function

# nice feature: factors protect you against invalid entries
my_factor[1] <- "z"

# removing factor
as.character(my_factor)
unclass(my_factor)
attributes(my_factor) <- NULL
my_factor

# more ways to create factors
my_factor <- factor(my_vector, levels = unique(my_vector))
my_factor
# the ordering in the levels, determines the order in the factor:
my_factor <- factor(my_vector, levels = letters)
my_factor[1] <- "z"

my_factor <- factor(my_vector, levels = rev(letters))
my_factor[1]
# unclass(my_factor)[1]

my_factor <- factor(my_vector, levels = letters, labels = rev(LETTERS))
my_factor <- factor(my_vector, levels = letters)
my_factor

# removing unobserved levels:
my_factor <- droplevels(my_factor)


# quirks and perks? factor is 'a kind of' container (advanced use of R!)
class(my_factor)

mode(my_factor) # wtf?
my_factor + 1 # pfew...
as.numeric(my_factor) # ok...
as.numeric(factor(my_factor, levels = rev(letters))) # you know why?
```

table | Session 5

```
N <- 3e4

my_numbers <- sample(rep(1:3, each= N / 10))
table(my_numbers)
my_chars <- sample(letters[1:3], 3e4, replace=T)
table(my_chars)

table(my_numbers, my_chars)

my_data <- data.frame(my_numbers, my_chars)
table(my_data)

my_logicals <- sample(c(TRUE, FALSE), N, replace=T)
table(my_numbers, my_chars, my_logicals)

my_data <- cbind(my_data, my_logicals)
table(my_data)
```

arrays | Session 6

```
my_array <- array(0, dim = c(2, 3))
my_array
class(my_array)

my_array <- array(1:24, dim = c(4, 3, 2))
# note, no byrow argument
# fills up by first, by second, by third, by ...
class(my_array)
my_array

my_array[1, 1 , 1]


my_array[, , 1]
my_array[1, , ]
my_array[, 1, ]

my_array <- array(1:120, dim = c(5, 4, 3, 2))
# pff...

# the table one was much easier right?
# everything is mode numeric
# turn it into proportions easily:
my_table <- table(my_data)
my_table/N


# apply:
apply(my_table, 3, sum) # row marginals
```

```r
apply(my_table, 1, sum) # depth(?) marginals
apply(my_table, c(1,2), sum) # row / column marginals (i.e. summing the two layers)
```