R Cheat Sheet: Writing Functions

```
Functions in R are called closures. Function environment
# Don't be deceived by the curly brackets: # When a function is called a new
                  # R is much more like Lisp than C or Java. # environment (frame) is created for it.
                 # Defining problems in terms of function # These frames are found in the call stack
# calls and their lazy, delayed evaluation # First frame is the global environment
                  # (variable resolution) is R's big feature. # Next fn reaches back into the call stack

Standard form (for named functions) called.by <- function() { # returns string
                            plus <- function(x, y) { x + y } # technically: who is my grandparent?
plus(5, 6) # -> 11 if(length(sys.parents()) <= 2)
# return() not needed – last value returned return('.GlobalEnv')
# Optional curly brackets with 1-line fns: deparse(sys.call(sys.parent(2)))
                      x.to.y <- function(x, y) return(x ^ y) } # Note: designed to be called from a fn
g <- function(...) { called.by() }
Returning values f <- function(...) g(...); f(a, 2)
# return() – can use to aid readability and
# for exit part way through a function Variable scope and unbound variables
# invisible() - return values that do not # Within a function, variables are
# print if not assigned. # resolved in the local frame first,
                   # Traps: return() is a function, not a # then in terms of super-functions (when a
                   # statement. The brackets are needed. # function is defined inside a function),
# then in terms of the global environment.
Anonymous functions h <- function(x) { x + a } # a undefined
# Often used in arguments to functions: a <- 5 # a defined in global environment
v <- 1:9; cube <- sapply(v,function(x) x^3) h(5) # -> returns 10
k <- function(x) { a <- 100; h(x) }
Arguments are passed by value k(10) # -> returns 15
# Effectively arguments are copied, and any # Note: local a in k() not seen in h()
               # changes made to the argument within the # variables not defined by the call stack!
# function do not affect the caller's copy. # [See my cheat sheet on R Environments]
# Trap: arguments are not typed and your
# function could be passed anything! Super assignment <<-
# Upfront argument checking advised! # x <<- y ignores the local x, and looks up
# the super-environments for a x to replace
Arguments passed by position or name accumulator <- function() {
                  b <- function(cat, dog, cow) cat+ dog+ cow a <- 0 # super assignment finds this a
b(1, 2, 3) # cat=1, dog=2, cow=3 function (x) {
                    b(cow=3, cat=1, dog=2) # order no problem a <<- a + x # the super assignment
                       b(co=3, d=2, ca=1) # unique abbreviations a # alone: this a will be printed
                  # Trap: not all arguments need be passed } # NOTE: anonymous function returned
f <- function(x) missing(x); f(); f('here') } # when accumulator() is called !!!
               # match.arg() – argument partial matching acc <- accumulator() # create accumulator
acc(1); acc(5); acc(2) # prints: 1, 6, 8
Default arguments
# Default arguments can be specified. Eg. Operator and replacement functions
                          x2y.1 <- function(x, y = 2) { x ^ y } `+`(4, 5) # -> 9 - operators are just fns
x2y.2 <- function(x, y = x) { x ^ y } `%plus%` <- function(a, b) { a + b }
                          x2y.2(3); x2y.2(2, 3) # -> 27 8 3 %plus% 2 # -> 5 # new defined functions
# "FUN(x) <- v is parsed as: x <- FUN(x, v)
The dots argument (...) is a catch-all "cap<-" <- function(x, value) # must use
                                     f <- function (...) { ifelse(x > value, value, x) # 'value'
# simple way to access dots arguments x <- c(1,10,100); cap(x) <- 9 # x -> 1,9,9
dots <- list(...) # return list
} Exceptions
                          x <- f(5); dput(x) # -> 5 (in a list) tryCatch(print('pass'), error=function(e)
g <- function (...) { print('bad'), finally=print('done'))
dots <- substitute(list(...))[-1] tryCatch(stop('fail'), error=function(e)
                          dots.names <- sapply(dots, deparse) print('bad'), finally=print('done'))
}
x <- g(a, b, c); dput(x)# -> c("a","b","c") Useful language reflection functions
# dots can be passed to another function: # exists(); get(); assign() – for variables
h <- function(x, ...) g(...) # substitute(); bquote(); eval(); do.call()
x <- h(a, b, c); dput(x) # -> c("b", "c") # parse(); deparse(); quote(); enquote()
```