R Cheat Sheet: Writing Functions

Functions in R are called closures.

```
# Don't be deceived by the curly brackets:
# R is much more like Lisp than C or Java.
# Defining problems in terms of function
# calls and their lazy, delayed evaluation
# (variable resolution) is R's big feature.
```

Standard form (for named functions)

```
plus <- function(x, y) { x + y }
plus(5, 6) # -> 11
# return() not needed - last value returned
# Optional curly brackets with 1-line fns:
x.to.y <- function(x, y) return(x ^ y)
```

Returning values

```
# return() - can use to aid readability and
#   for exit part way through a function
# invisible() - return values that do not
#   print if not assigned.
# Traps: return() is a function, not a
#   statement. The brackets are needed.
```

Anonymous functions

```
# Often used in arguments to functions:
v <- 1:9; cube <- sapply(v,function(x) x^3)
```

Arguments are passed by value

```
# Effectively arguments are copied, and any
# changes made to the argument within the
# function do not affect the caller's copy.
# Trap: arguments are not typed and your
#   function could be passed anything!
#   Upfront argument checking advised!
```

Arguments passed by position or name

```
b <- function(cat, dog, cow) cat+ dog+ cow
b(1, 2, 3) # cat=1, dog=2, cow=3
b(cow=3, cat=1, dog=2) # order no problem
b(co=3, d=2, ca=1) # unique abbreviations
# Trap: not all arguments need be passed
f <- function(x) missing(x); f(); f('here')
# match.arg() - argument partial matching
```

Default arguments

```
# Default arguments can be specified. Eg.
x2y.1 <- function(x, y = 2) { x ^ y }
x2y.2 <- function(x, y = x) { x ^ y }
x2y.2(3); x2y.2(2, 3) # -> 27 8
```

The dots argument (...) is a catch-all

```
f <- function (...) {
    # simple way to access dots arguments
    dots <- list(...) # return list
}
x <- f(5); dput(x) # -> 5 (in a list)
g <- function (...) {
    dots <- substitute(list(...))[-1]
    dots.names <- sapply(dots, deparse)
}
x <- g(a, b, c); dput(x)# -> c("a","b","c")
# dots can be passed to another function:
h <- function(x, ...) g(...)
x <- h(a, b, c); dput(x) # -> c("b", "c")
```

Function environment

```
# When a function is called a new
#   environment (frame) is created for it.
# These frames are found in the call stack
# First frame is the global environment
# Next fn reaches back into the call stack

called.by <- function() { # returns string
    # technically: who is my grandparent?
    if(length(sys.parents()) <= 2)
        return('.GlobalEnv')
    deparse(sys.call(sys.parent(2)))
} # Note: designed to be called from a fn
g <- function(...) { called.by() }
f <- function(...) g(...); f(a, 2)
```

Variable scope and unbound variables

```
# Within a function, variables are
# resolved in the local frame first,
# then in terms of super-functions (when a
# function is defined inside a function),
# then in terms of the global environment.
h <- function(x) { x + a } # a undefined
a <- 5 # a defined in global environment
h(5) # -> returns 10
k <- function(x) { a <- 100; h(x) }
k(10) # -> returns 15
# Note: local a in k() not seen in h()
# variables not defined by the call stack!
# [See my cheat sheet on R Environments]
```

Super assignment <<-

```
# x <<- y ignores the local x, and looks up
# the super-environments for a x to replace
accumulator <- function() {
    a <- 0 # super assignment finds this a
    function (x) {
        a <<- a + x # the super assignment
        a # alone: this a will be printed
    } # NOTE: anonymous function returned
}     # when accumulator() is called !!!
acc <- accumulator() # create accumulator
acc(1); acc(5); acc(2) # prints: 1, 6, 8
```

Operator and replacement functions

```
`+`(4, 5) # -> 9 - operators are just fns
`%plus%` <- function(a, b) { a + b }
3 %plus% 2 # -> 5 # new defined functions
# "FUN(x) <- v is parsed as: x <- FUN(x, v)
"cap<-" <- function(x, value)   # must use
    ifelse(x > value, value, x) # 'value'
x <- c(1,10,100); cap(x) <- 9 # x -> 1,9,9
```

Exceptions

```
tryCatch(print('pass'), error=function(e)
  print('bad'), finally=print('done'))
tryCatch(stop('fail'), error=function(e)
  print('bad'), finally=print('done'))
```

Useful language reflection functions

```
# exists(); get(); assign() - for variables
# substitute(); bquote(); eval(); do.call()
# parse(); deparse(); quote(); enquote()
```

