1. Writing functions

A *function* (or *procedure* or *subprogram*) includes a line describing its input *arguments* and a *body* consisting of one or several statements.

We know many functions, e.g. `c()`, `sum()`, and `read.csv()`, which calls `read.table()`, whose code is hard to write and read (run `read.table` to see it) but easy to use.

Writing a function facilitates:

- *abstraction*, by which we focus on relevant information while ignoring implementation details;
- *top-down design*, which breaks a task into easier subtasks, each solved by its own function;
- *clarity* of code, since it is easier to read a function name than the code it summarizes;
- *correctness* of code, since it is easy to test, debug, and optimize a short function; and
- *code reuse*, since we write, test, and debug a function once but call it many times.

Designing and writing functions well is essential to good programming.

Here is the format of an R function definition (where `UPPER.CASE` text is a placeholder for R code):

```
FUNCTION.NAME = function(PARAMETER.LIST) {
  BODY
}
```

e.g.

```
letterhead = function() { # define new function "letterhead" with no arguments
  cat(sep="", "1300 University Avenue\n")
  cat(sep="", "Madison WI 53706\n")
}

standardize = function(x, mu=0, sigma=1) { # define "standardize" with 3 args.
  z = (x - mu) / sigma
  return(z)
}

z = 3
letterhead()                                  # call letterhead()
standardize(x=10, mu=6, sigma=2)              # call standardize()
z                                             # standardize() did not change z
z = standardize(4, mu=6)                      # now z changes
z
z = letterhead()                              # letterhead() returns NULL
z
```

A function call proceeds as follows:

- Execution jumps to first line of function upon seeing the call, `FUNCTION.NAME(ARGUMENT.LIST)`
- Function's `PARAMETER.LIST` is *copied* from caller's `ARGUMENT.LIST` by name or position, and from defaults specified as `PARAMETER.NAME=DEFAULT` in `PARAMETER.LIST`
- Assignment to function parameters and local variables doesn't affect caller's variables
- Code in function is executed until `return(EXPRESSION)`, or until function's closing `}`
- Execution returns to caller; if caller assigned a variable to function, it gets `EXPRESSION` from function's `return()` or last expression

  Note: `return()` and `cat()` are not the same thing. `return()` returns a value to which the caller can assign a variable, which affects the state of the program. `cat()` (like `print()`) writes text on the console, which can be helpful to a human reader, but it doesn't affect the state of the program. Most functions should use `return()`, not `cat()`, to provide output.

A note on testing numeric functions

Here's an idiom to check whether a function call `f(ARGUMENT.LIST)` returns the expected `answer`:

```
stopifnot(isTRUE(all.equal(answer, f(ARGUMENT.LIST))))
```

e.g. `stopifnot(isTRUE(all.equal(2, standardize(x=10, mu=6, sigma=2))))`

Background:

- Using `==` to test equality between real numbers (`numeric` in R) is unreliable. e.g.

  ```
  49*(1/49) == 1
  ```

- `all.equal(x, y, tolerance=(.Machine$double.eps^0.5))` returns `TRUE` if `x` is close to `y` (that is, if `abs(x - y) < tolerance`). Otherwise it returns an error message. e.g.

  ```
  all.equal(49*(1/49), 1)
  ```

  ```
  all.equal(0, 1) # it would be convenient if this were FALSE
  ```

- `isTRUE(x)` returns `TRUE` if `x` is `TRUE`, and `FALSE` otherwise.
- `stopifnot(...)` stops if any logical expression in `...` is not all TRUE. e.g.

  ```
  stopifnot(0 < 1)
  ```

  ```
  stopifnot(0 < 1, 1:3 < 3)
  ```

A more interesting example of a user-defined function

```
# Description: baby.t.test runs a one-sample t-test and finds a confidence
#   interval for an unknown mean. (It implements part of R's t.test().)
# Usage: baby.t.test(x, mu=0, conf.level=.95)
# Parameters:
#   x: a numeric vector of data values (must have at least two values)
#   mu: the hypothesised true mean
#   conf.level: confidence level of the interval (must be in (0, 1))
# Details: the test is for the null hypothesis that the true mean is
#   mu against the alternative that the true mean is not mu
# Value: a list containing these components:
#   $statistic: the t statistic
#   $parameter: degrees of freedom for the t statistic
#   $p.value: probability of a t statistic more extreme than the one computed
#   $conf.int: a confidence interval for the true mean
#   $estimate: the estimated (sample) mean
#   $null.value: the specifited hypothesized value of the mean (mu)
baby.t.test = function(x, mu=0, conf.level=.95) {
  stopifnot(length(x) >= 2)
  stopifnot((0 < conf.level) & (conf.level < 1))
  n = length(x)
  x.bar = mean(x)
  s.x = sd(x)
  t = (x.bar - mu) / (s.x / sqrt(n))
  r = list() # set up r as return value
  r$statistic = t
  r$parameter = n - 1
  r$p.value = 2*pt(q=-abs(t), df=n-1)
  alpha = 1 - conf.level
  t.for.conf.level = -qt(p=alpha/2, df=n-1)
  error.margin = t.for.conf.level * s.x / sqrt(n)
  r$conf.int = c(x.bar - error.margin, x.bar + error.margin)
  r$estimate = x.bar
  r$null.value = mu
  return(r)
}
baby.t = baby.t.test(1:10, 5, .90) # compare my function to real t.test()
t = t.test(x=1:10, mu=5, conf.level=.90) # as.numeric(), below, strips names
stopifnot(isTRUE(all.equal(baby.t$statistic, as.numeric(t$statistic))))
stopifnot(isTRUE(all.equal(baby.t$parameter, as.numeric(t$parameter))))
stopifnot(isTRUE(all.equal(baby.t$p.value, as.numeric(t$p.value))))
stopifnot(isTRUE(all.equal(baby.t$conf.int, as.numeric(t$conf.int))))
stopifnot(isTRUE(all.equal(baby.t$estimate, as.numeric(t$estimate))))
stopifnot(isTRUE(all.equal(baby.t$null.value, as.numeric(t$null.value))))
```