MATH 3080 Lab Lecture 1

Curtis Miller

12/05/2019

Lecture 1

Control Flow

In programming, **control flow** is the order in which statements in a program are evaluated, if they're evaluated at all. R is a full-featured programming language and thus allows for control flow statements. We will review those statements here.

`if` and `else`

`if` statements allows for conditional evaluation of code, and take the form `if (condition) {code}`.[1] `condition` must be a statement that evaluates to a single `TRUE` or `FALSE`. If `condition` is `TRUE`, then `code` will be run. However, programmers may want certain code to run if `condition` is `FALSE`; for this purpose, `else` exists, and we may use it in the format `if (condition) {code_true} else {code_false}`. In fact, programmers may want to check a series of possibilities and execute code based on which is true, in which case we can write `else if` to add more contingencies to our `if` statement.

Below are some examples of using `if` statements.

```r
if (1 + 1 == 2) {
    print("Arithmetic works!")
}
```

```
## [1] "Arithmetic works!"
```

```r
if (0 < 1) {
    print("We have order!")
} else {
    print("We don't have order!")
}
```

```
## [1] "We have order!"
```

```r
x <- 0

if (x < 0) {
    print("x is less than zero")
} else if (x == 0) {
    print("x is zero")
} else {
    print("x is greater than zero")
}
```

[1] When on a single line, the braces `{}` are actually optional, but I recommend always using them for style purposes; your code is more robust and easily changed if you always use braces.

```
## [1] "x is zero"
```

Similar to `if` and `else` is the `ifelse()` function, used like so: `ifelse(vec, return_true, return_false)`. `ifelse()` is a vectorized function; `vec` could be a vector consisting of `TRUE` and `FALSE` value. For every `TRUE` in `vec`, `return_true` will be returned; elsewhere, `return_false` is returned.

```r
(x <- rnorm(10))
```

```
##  [1]  2.36322563  0.30047965 -1.89357579  0.26239552 -0.05683042
##  [6] -0.64985772  0.18922230 -0.56059260  0.26401172  0.49756469
```

```r
# We will return a vector where all negative numbers are zero; otherwise, the
# original value of the vector is kept.

x < 0
```

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

```r
ifelse(x < 0, 0, x)  # Put 0 where TRUE, else put corresponding x value
```

```
##  [1] 2.3632256 0.3004796 0.0000000 0.2623955 0.0000000 0.0000000 0.1892223
##  [8] 0.0000000 0.2640117 0.4975647
```

`switch()`

Suppose you wanted a value to depend on some input, and many different values for that input are possible. For example, a string determines the statistic to return. We could chain `if` and `else` statements like so:

```r
stat <- "mean"
x <- rnorm(10)

if (stat == "mean") {
    mean(x)
} else if (stat == "median") {
    median(x)
} else if (stat == "max") {
    max(x)
} else if (stat == "min") {
    min(x)
}
```

```
## [1] 0.02902361
```

The above code works but is verbose and error-prone[2]. We could tidy our code by using `switch()` which allows for passing parameters that can provide multiple outputs for multiple cases.

Here's the above code block tidied by `switch()`:

```r
switch(stat,
  mean = mean(x),
  median = median(x),
  max = max(x),
  min = min(x)
)
```

```
## [1] 0.02902361
```

[2]The above code could be quickly written and vastly improved like so: `stat <- get("mean"); stat(x)`. Try it! What just happened? What does `get()` do? Without using `get()` or a string, we could similarly try `stat <- mean; stat(x)`. Either solution is probably better than using `switch()`, though.

`tryCatch()`

`tryCatch()` is a function that allows for conditionally running code depending on whether an error appeared or not. Rather than a program simply failing when an error occurs, `tryCatch()` facilitates nuanced approaches to error handling. A common syntax for `tryCatch()` is `tryCatch(expr, error = function(e) {do_something()})`. `expr` is some code to be run, and the function passed to the `error` argument handles the error object `e`. An optional parameter, `finally`, allows for an expression that will always be run regardless of whether there was an error or not, and will be evaluated before the result of `expr` is returned.

```
tryCatch(1 + 1, error = function(e) {"Help!"}, finally = print("Hello!"))
```

```
## [1] "Hello!"
```

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

```
tryCatch(1 + "a", error = function(e) {"Help!"}, finally = print("Hello!"))
```

```
## [1] "Hello!"
```

```
## [1] "Help!"
```

```
# Errors are actually objects that we can inspect; here I capture this object
error_obj <- tryCatch(1 + "a", error = function(e) {e})
str(error_obj)
```

```
## List of 2
##  $ message: chr "non-numeric argument to binary operator"
##  $ call   : language 1 + "a"
##  - attr(*, "class")= chr [1:3] "simpleError" "error" "condition"
```

```
error_obj$message
```

```
## [1] "non-numeric argument to binary operator"
```

Repeated Evaluation

A **loop** is a section of code evaluated repeatedly before a program proceeds to later code. All loops consist of a body of code to be repeated and a condition determining if the loop needs to be terminated. **Beware:** if this condition never occurs, the loop will never end and the program will never stop unless some outside force (such as a kill signal from the operating system) terminates the program.[3]

A `while` loop may be the simplest loop; the body is run until the condition passed to the loop becomes false. These loops take the form `while (condition) {code}`.

```
x <- 0
while (x < 10) {
    print(x)
    x <- x + 1
}
```

```
## [1] 0
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
```

[3]Actually many programs consist of loops that basically never end. Video games and graphical programs, for example, are understood as operating in a loop that will not terminate until the whole program terminates.

```
## [1] 8
## [1] 9
```

Below is a simple loop that never ends:

```
while (TRUE) {
    print("I'm a potatoe!")
}
```

`for` loops use list-like objects in their condition and also have a variable that represents each element in the list. The syntax for `for` loops is `for (var in l) {do_something_with(var)}`. These loops can be viewed as `while` loops where the condition for continuation is that there are more elements in the list to process.

```
x <- 1:10
for (i in x) {
    print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
```

```
# A Fibonacci number calculator
m <- 10  # Tenth number in Fibonacci sequence
f_num <- 1  # The first number in the sequence
s_num <- 1  # The second number in the sequence
for (i in 1:(m - 2)) {
  t_num <- f_num + s_num
  f_num <- s_num
  s_num <- t_num
}
t_num
```

```
## [1] 55
```

```
# Equivalent while loop
f_num <- 1
s_num <- 1
l <- 1:(m - 2)
n <- length(l)
idx <- 1
while (idx <= n) {
  i <- l[idx]
  t_num <- f_num + s_num
  f_num <- s_num
  s_num <- t_num
  idx <- idx + 1
}
t_num
```

```
## [1] 55
```

The expression `break` when used in a loop allows for early termination of the loop.

```
x <- 0
while (TRUE) {  # Will this loop end?
    x <- x + 1
    if (x > 10) {
        print("Potatoe!")
        break
    }
}
```

```
## [1] "Potatoe!"
```

`repeat` is essentially a `while` loop with a condition that's always true; the only way to break the loop (other than a kill signal) is via `break`.

```
# This loop is equivalent to the loop above
x <- 0
repeat {
  x <- x + 1
  if (x > 10) {
    print("Potatoe!")
    break
  }
}
```

```
## [1] "Potatoe!"
```

You may have heard never to use loops such as `for` loops when programming in R. There are good reasons to avoid `for` loops; you should not use them in place of vectorized operations, for example. However, there are times when loops are unavoidable (the Fibonacci sequence is an example of a sequence that's hard to handle without loops) and when programming one should be ready to use any necessary loops.

Throwing Errors and Warnings

While novice programmers strongly dislike errors they are important for programming. Ample errors and warnings allow for easier debugging and prevent programs from entering unwanted territory; an error thrown near the initial bad code can save hours of time trying to find the initial problem. Thus programmers look for places to throw errors and warnings to make sure that programs run as expected and don't behave badly.

In R there are three classes of run-time messaging that can be used for behavior management; ranked from most to least severe, there are **errors**, **warnings**, and **messages**. We can throw an error using the function `stop()`, where the input to `stop()` is the message to accompany the error.

```
x <- "a"
if (is.character(x)) {
  stop("x should not be a character")
} else {
  x + 1
}
```

```
## Error in eval(expr, envir, enclos): x should not be a character
```

Errors should be thrown via `stop()` when the appropriate course of action is for the program to stop and not proceed. This contrasts with throwing a warning, where the program can still proceed but the user should be notified that some belief about the state of the program has been violated.

```
u <- 1
if (!is.logical(u)) {
```

```r
  warning("u is not boolean!")
}
```

```
## Warning: u is not boolean!
```

```r
u | FALSE
```

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

Finally there are messages. Messages simply alert the user to the current operation of the program; they do not necessarily indicate “bad” behavior. Packages, for example, produce messages as they load.

```r
sum <- 0
for (i in 1:10) {
  message(i)
  sum <- sum + i
}
```

```
## 1

## 2

## 3

## 4

## 5

## 6

## 7

## 8

## 9

## 10
```

```r
sum
```

```
## [1] 55
```

Functions

You should already have seen functions and function authoring before, so consider this sentence review: a **function** is a structure in a program that will execute some segment of code when called. Functions often take inputs and will return outputs depending on those inputs. In R, functions are objects that can be created like any other data object; to repeat, functions in R can be treated as data. This means that:

- You can save function in variables, vectors, lists, etc. (For example, the following is legal and perfectly reasonable code: `c(mean, median, sd)`.)
- Functions can be passed to functions as arguments; we call functions accepting functions as inputs **functionals**. (An example of functionals are the `apply()` collection of functions, including `lapply()`.)
- Functions can be *returned* by functions, as an output; we call the function returned by another function a **closure**. (An example of a function that produces closures is `Vectorize()`.)

Functions consist of three key ingredients: the **formals**, the **body**, and the **environment**. Formals are arguments the function accepts. The function body is the block of code executed by the function when called. The environment is the data structure in which the function is defined; user functions generally are defined in the global environment, but closures often live in a different environment.

The following illustration demonstrates the relationship between these three things:

```
# This function was defined by the user and thus lives in the global environment

function(formals) {
  body
}
```

There are functions that pick a function apart into these component parts:

```
increment <- function(x) {
    x + 1
}
formals(increment)
```

```
## $x
```

```
body(increment)
```

```
## {
##     x + 1
## }
```

```
environment(increment)
```

```
## <environment: R_GlobalEnv>
```

The function `args()` is primarily interested in the formals of a function, but for interactive use, showing R users what arguments a function takes and their default values. `args()` should only be used interactively to personally learn about a funciton, not for more advanced programming; when programming, use `formals()`.

```
args(paste)
```

```
## function (..., sep = " ", collapse = NULL)
## NULL
```

```
formals(paste)  # The result is a list with named elements, and entries of the
```

```
## $...
##
##
## $sep
## [1] " "
##
## $collapse
## NULL
```

```
                # list being parameter defaults
```

Infix Notation

Recall the rules for function names; syntactically valid names include alphanumeric characters, `.`, and `_`, but cannot start with numbers or `_` (so `.Mean_1_` is a valid name, but `1.Mean_` and `_Mean.1_` are not). However, we can give objects syntactically invalid names by using backquotes, like so:

```
.Mean_1_ <- 1          # Syntactically valid name
`1.Mean_` <- 2         # Invalid name
`_Mean.1_` <- 3        # Invalid name
`Awesome sauce!` <- 4  # Invalid name

.Mean_1_
```

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

```r
`.Mean_1_`
```

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

```r
`1.Mean_`
```

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

```r
`_Mean.1_`
```

```
## [1] 3
```

```r
`Awesome sauce!`
```

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

One particular case where we may want syntactically invalid names is to define infix functions. You've already met one: `+`. Yes, `+` is a function, as demonstrated below:

```r
`+`
```

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

```r
class(`+`)
```

```
## [1] "function"
```

```r
`+`(1, 2)
```

```
## [1] 3
```

In short, an infix function is a function `f` that can be called with two arguments like so: `x f y`. Most user-defined infix functions, though, need to have their name wrapped by two `%` symbols, like `%my_function%`.

R does come with some such function already defined (excluding "trivial" ones like `+`). `%*%` computes matrix (inner) products (the product of two matrices as taught in linear algebra, as opposed to element-wise products as done by `*`), and `%o%` matrix outer products. The **dplyr** operator `%>%` is another example. These are not the only useful infix operators we could imagine, though.

For example, R does not have an operator for string concatenation (meaning combining two strings; when we concatenate `"string1"` and `"string2"`, we get the string `"string1string2"`). We can define an infix operator for string concatenation like so:

```r
`%s%` <- function(x, y) {paste(x, y)}    # Concatenate, separating with space
`%s0%` <- function(x, y) {paste0(x, y)}  # Concatenate, no separating characters

"hello" %s% "world"
```

```
## [1] "hello world"
```

```r
"hello" %s0% "world"
```

```
## [1] "helloworld"
```

Variadic Arguments

When looking at some function arguments, you may on occasion notice the argument `...`. This argument actually refers to a list of arguments of undefined length. This allows for writing functions that can take arguments not necessarily defined outright by the programmer in the function definition.

We allow for variadic arguments when we include `...` in the function definition, and use them by referring to `...` in a function call. Any argument passed to the function that was not named as an argument to the function is included in `...`.

Below is a function that collects some arguments in `...` and attempts to return them in a vector (this could be problematic if not all the arguments are of the same type; I would recommend collecting in a list instead in general). The function also has a named argument that gets printed.

```r
collector <- function(..., stringout = "Hello!") {
    print(stringout)
    vec <- c(...)
    return(vec)
}

u <- collector(1, 1, 4)
```

```
## [1] "Hello!"
```

```r
u
```

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

```r
u <- collector(a = 1, b = 1, c = 1, stringout = 10)
```

```
## [1] 10
```

```r
u
```

```
## a b c
## 1 1 1
```

One can place `...` anywhere in the funciton defintion, but beware: where it's placed matters. Named arguments prior to `...` are treated as positional arguments, and arguments after `...` must be referred to explicitly if they are to be modified.

```r
new_collector <- function(introstring = "Hello!", ...,
                          leavestring = "Goodbye!") {
  print(introstring)
  u <- c(...)
  print(leavestring)

  u  # u will be returned when referred to like this
}

x <- new_collector("Whoa", "Awesome", "world")
```

```
## [1] "Whoa"
## [1] "Goodbye!"
```

```r
x
```

```
## [1] "Awesome" "world"
```

```r
x <- new_collector(1, 2, 3, 4)
```

```
## [1] 1
## [1] "Goodbye!"
```

```r
x
```

```
## [1] 2 3 4
```

```r
x <- new_collector("whoa", "awesome", "stuff", leavestring = "solo")
```

```
## [1] "whoa"
## [1] "solo"
```

```
x
```

```
## [1] "awesome" "stuff"
```

```
x <- new_collector(1, 2, 3, introstring = "whoa", leavestring = "awesome")
```

```
## [1] "whoa"
## [1] "awesome"
```

```
x
```

```
## [1] 1 2 3
```

So far I have demonstrated `...` by immediately passing it to `c()`, but in fact we can pass `...` to any function. Let's demonstrate by writing our own version of the function `paste()`. The argument `sep` controls the separator between words in the string. We could set `sep = "_"` to separate with _ rather than a space (the default). That said, the *only* parameter we want to change is `sep`; we otherwise want our new function to behave exactly like `paste()`, and we don't want to have to rewrite `paste()` to accomodate this. Also, we prefer simple code; we don't want to have to repeat all of `paste()`'s arguments in our function definition, especially since we want our new function to work with all past and future versions of `paste()` that could behave differently in other versions of R. (Actually I don't expect the `paste()` syntax to ever change, but the moral of the story stands.)

Given these design constraints, `...` is extremely useful, since we only need to write the following:

```
paste_ <- function(..., sep = "_") {
    paste(..., sep = sep)
}
paste("hello", "world")
```

```
## [1] "hello world"
```

```
paste_("hello", "world")
```

```
## [1] "hello_world"
```

```
paste_("source", 1:10)
```

```
##  [1] "source_1"  "source_2"  "source_3"  "source_4"  "source_5"
##  [6] "source_6"  "source_7"  "source_8"  "source_9"  "source_10"
```

```
paste_("source", 1:10, collapse = "...")             # A paste() argument
```

```
## [1] "source_1...source_2...source_3...source_4...source_5...source_6...source_7...source_8...source_
```

```
paste_("source", 1:10, collapse = "...", sep = "-")  # Stupid but valid
```

```
## [1] "source-1...source-2...source-3...source-4...source-5...source-6...source-7...source-8...source-
```

Demonstration: z Statistic Function

Recall from previous courses the z statistic:

$$z = \frac{\bar{x} - \mu_0}{\frac{\sigma}{\sqrt{n}}}$$

Here, $\bar{x}$ is the sample mean, n the sample size, μ_0 is the mean of the data under the null hypothesis, and σ is the population standard deviation. Let's write a function that computes the z statistic. How will our function work? We will sketch the following specification:

- We will call our function `z.stat()`.

- `z.stat()` will take a list of numbers *directly* and compute the z statistic. (This is probably not wise from a design perspective but allows us to demonstrate the use of `...`.)
- Two named parameters for `z.stat()` will be `sigma` and `mu`, representing the population standard deviation and mean under the null hypothesis, respectively
- The function should raise an error if the input data is not numeric or if `sigma` is not positive. We want the function to check and throw its own errors in these cases; while we might consider letting the arithmetic throw the errors, having the function throw the error allows for easier error diagnosis should an unwitting user make a mistake.
- We want the function to work for boolean (i.e. `TRUE`/`FALSE`) data but throw a warning; while boolean can be easily converted to numeric, we don't want our function used this way, though we don't want to explicitly ban this usage.
- The user does not absolutely have to specify `sigma`; our function could fall back on the sample standard deviation in that case. However, we don't want to encourage this use, so we should throw a warning if this occurs.

(I'm not going to argue this is good design; this is more about demonstrating techniques than using them well. That said, writing a design document for code is a very good idea, and essential if the code is going to be distributed or if a team is working on it.)

```r
# The following comments are more than just comments; they're ready for
# interpretation by a package called roxygen2, which can create documentation
# for functions and other package objects.

#' Z Statistic
#'
#' Compute the \eqn{z}-statistic for a given data set.
#'
#' @param ... Data for which to compute the \eqn{z}-statistic
#' @param mu The mean of the population under the null hypothesis
#' @param sigma The population standard deviation; unset by default (this is
#'              achieved by making the default value of the parameter
#'              \code{NULL})
#' @return The \eqn{z}-statistic
#' @examples
#' z.stat(1, 2, 3, mu = 2, sigma = 1)
z.stat <- function(..., mu = 0, sigma = NULL) {
  x <- tryCatch(c(...),  # Attempt to convert ... into a vector
      error = function(e) {
        "Inappropriate data passed to z.stat(); it should be numeric"
      })
  if (is.logical(x)) {
    warning("Data passed to z.stat() is logical; may be inappropriate for z" %s%
            "test")
  } else if (!is.numeric(x)) {
    stop("Data passed to z.stat() must be numeric")
  }

  if (is.null(sigma)) {
    warning("sigma not set; defaulting to sample standard deviation, but" %s%
            "statistic may not be appropriate")
    sigma <- sd(x)
  }

  if (!is.numeric(sigma) | sigma <= 0) {
    stop("sigma must be a positive number")
  }

  xbar <- mean(x)
  n <- length(x)

  (xbar - mu) / (sigma / sqrt(n))
}
```

Let's run some tests to see that our function works as intended.

```r
z.stat(0.1, 1.1, -0.1, -2.5, 0.3)  # Produces a warning
```

```
## Warning in z.stat(0.1, 1.1, -0.1, -2.5, 0.3): sigma not set; defaulting to
## sample standard deviation, but statistic may not be appropriate
```

```
## [1] -0.3634502
```

```r
z.stat(0.1, 1.1, -0.1, -2.5, 0.3, sigma = 1)
```

```
## [1] -0.491935
```

```r
z.stat(0.1, 1.1, -0.1, -2.5, 0.3, mu = 1, sigma = 1)
```

```
## [1] -2.728003
```

```r
z.stat("0.1", 1.1, -0.1, -2.5, 0.3, mu = 1, sigma = 1)  # Should produce error
```

```
## Error in z.stat("0.1", 1.1, -0.1, -2.5, 0.3, mu = 1, sigma = 1): Data passed to z.stat() must be nume
```

```r
z.stat(TRUE, TRUE, FALSE, TRUE, FALSE, mu = 1, sigma = 1)  # Produces warning
```

```
## Warning in z.stat(TRUE, TRUE, FALSE, TRUE, FALSE, mu = 1, sigma = 1): Data
## passed to z.stat() is logical; may be inappropriate for z test
```

```
## [1] -0.8944272
```

```r
z.stat(0.1, 1.1, -0.1, -2.5, 0.3, mu = 1, sigma = -1)  # Produces error
```

```
## Error in z.stat(0.1, 1.1, -0.1, -2.5, 0.3, mu = 1, sigma = -1): sigma must be a positive number
```

Based on our tests our function seems to work appropriately.