R Programming Week 2

Rudy Rupp

13/10/2020

R Programming Week 2: Programming with R

Learning Objectives

- write if-else expressions.
- write "for", "while" and "repeat" loops
- define a function and specify its return value
- describe how R binds a value to a symbol via the search list
- explain lexical scoping with respect to how the value of free variables are resolved.
- describe the difference between lexical scoping and dynamic scoping rules.
- convert character strings representing date & time to an R datetime object.

Control Structures

allow you to control the flow of execution of a program. For example:

- *if*, *else* tests a condition.
- *for* repeat a fixed number of times.
- *while* repeat while a condition is true.
- *repeat* repeats indefinitely.
- *break* stops the execution of a loop.
- *next* skips one iterations.
- *return* exists a function.

These are used when writing programs. When writing on the command line, there are a separate set of functions with *apply* which are more useful for loops.

if, else

The "else" argument is not necessary.

```
x <- rnorm(1, mean = 3)
x
```

```
## [1] 3.019035
```

```
if(x>3) {
        y <- 10
} else {
        y <- 0
}
y
```

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

Can also be written as:

```
x <- rnorm(1, mean = 3)
x
```

```
## [1] 3.412213
```

```
y <- if(x>3) {
             10
} else {
             0
}
y
```

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

"For" loops

for loops take an interator variable and assign it successive values from a sequence of a vector. Most commonly used for iterating over the elements of an object.

```
for(i in 1:10) {
                print(i*2)
}  ## a simple example
```

```
## [1] 2
## [1] 4
## [1] 6
## [1] 8
## [1] 10
## [1] 12
## [1] 14
## [1] 16
## [1] 18
## [1] 20
```

```
x <- c("a", "b", "c", "d", "e")
for (i in 1:4) {
                print(x[i])
  }  ## subsetting x with i to give the first 4 elements of x
```

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

```
for(i in seq_along(x)) {
                        print(x[i])
}  ## seq_along gives all the elements
```

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

```
for(letter in x) {
                  print(letter)
}  ## here "letter" is just an object. Gives all values.
```

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

```
for(i in 1:4) print(x[i]) ## in the same line, the "{" is not needed.
```

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

They can also be nested:

```
x <- matrix(1:6, 2, 3)
x
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```
for(i in seq_len(nrow(x))) {
     ## seq_len creates an integer sequence up to the length of the
     ## object. In this case, the number of rows of x, so the
```

```
     ## sequence is 1 2.
          for(j in seq_len(ncol(x))) {
            ## here seq_len will create the sequence 1 2 3,
            ## as there are 3 columns.
                  print(x[i, j])
            ## as i and j change, here they are the same as the row
            ## and column from the matrix. the subsetting will
            ## extract only one element.
          }
}
```

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

Nesting loops beyond 2 or 3 levels can be very difficult to read.

"While" Loops

The loop begins by testing a condition, and if it's true, runs the code. It then tests the condition again before each iteration.

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

```
## [1] 0
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
```

Can become infinite if not well written. Be careful.

There can be more than one condition in the test

```
z <- 5
while(z >= 3 && z <= 10) {
          coin <- rbinom(1, 1, 0.5)  ## set to 1 or 0 randomly
          print(c(z, coin)) ## print value and coin result
          if(coin == 1) {
                    z <- z + 1
```

```
          } else {
                    z <- z - 1
          }     ## step up or down at random until a value is reached
                ## that doesn't meet the conditions.
}
```

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

Conditions are evaluated left to right.

"Repeat" Loops, "Next" and "Break"

Repeat is not frequently used because it continues to run forever, or until *break* is called within it (typically inside an if function).

```
x0 <- 10000
tol <- 1

repeat {
        x1 <- sample(-x0:x0, 1)
        print(c("x is ", x1), quote = FALSE)
        if(abs(x1 - x0) <= tol) {
           ## breaks the loop in the two values are within a
           ## tolerance limit set as tol
                break
        } else {
        x0 <- x1
        }
}
```

```
## [1] x is  -4445
## [1] x is  899
## [1] x is  -489
## [1] x is  -12
## [1] x is  -2
## [1] x is  0
## [1] x is  0
```

It's safer to use a *for* loop with many iterations, then report whether a break was achieved or not.

```
x0 <- 10000
tol <- 1
conv <- "Convergeance has NOT been achieved"

for(i in 1:10) {
        x1 <- sample(-x0:x0, 1)
        print(c("x is ", x1), quote = FALSE)
        if(abs(x1 - x0) <= tol) {
                print(c(i, "iterations"), quote = FALSE)
                conv <- "Convergeance HAS been achieved"
                break
        } else {
        x0 <- x1
        }
}
```

```
## [1] x is  -3316
## [1] x is  -1280
## [1] x is  643
## [1] x is  -83
## [1] x is  62
## [1] x is  -59
## [1] x is  -3
## [1] x is  -1
## [1] x is  0
## [1] 9          iterations
```

```
print(conv)
```

```
## [1] "Convergeance HAS been achieved"
```

This way, if there's a problem with the code, the program will still finish and give you an end. *return* can also finish a loop, but also finishes any functions. Will return to this argument later.

Next can skip one iteration of a loop. E.g.:

```
for(i in 1:30) {
            if(i <= 10) {
                    next
                    ## skips first 10 iterations
            }
            print(i)
}
```

```
## [1] 11
## [1] 12
## [1] 13
## [1] 14
## [1] 15
## [1] 16
## [1] 17
## [1] 18
```

```
## [1] 19
## [1] 20
## [1] 21
## [1] 22
## [1] 23
## [1] 24
## [1] 25
## [1] 26
## [1] 27
## [1] 28
## [1] 29
## [1] 30
```

Functions

Functions are typically held in a separate text file.

```
source("My_Functions.R")
```

```
## [1] "Code read"
```

```
## a simple function that adds 2 numbers together
add2(286, 365)
```

```
## [1] 651
```

```
## a simple function that gives all elements of a vector that are
## higher than a threshold. Default threshold is 10.
above(1:20)
```

```
##  [1] 11 12 13 14 15 16 17 18 19 20
```

```
## This function gives the mean value of each column in a matrix
## or data frame. NAs are removed by default, but this can be turned
## off
columnmean(airquality)
```

```
## [1]  42.129310 185.931507   9.957516  77.882353   6.993464  15.803922
```

Functions are stored as R objects, with class "function". They are "first class objects", which means they can be treated similarly to other objects. They can be passed as arguments in other functions and they can be nested.
The return value of a function is the last expression in its body to be evaluated.

Not all arguments of a function must be included. They may have default values, and they may be called by name or position. E.g.:

```
mydata <- rnorm(100)
## consider the sd function, that calculates standard deviations.
sd(mydata)
```

```
## [1] 1.009112
```

```r
sd(x = mydata)
```

```
## [1] 1.009112
```

```r
sd(x = mydata, na.rm = FALSE)
```

```
## [1] 1.009112
```

```r
sd(na.rm = FALSE, x = mydata)
```

```
## [1] 1.009112
```

```r
sd(na.rm = FALSE, mydata)
```

```
## [1] 1.009112
```

```r
   ## in th last example, the order is changed. it might work but
   ## is not recommended.
```

Positional matching and matching by name can be mixed, which is useful for long list of arguments. E.g.:

```r
mydata <- data.frame(x = 1:50, y = sequence(50, from = 27, by = 2))

args(lm)
```

```
## function (formula, data, subset, weights, na.action, method = "qr",
##     model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE,
##     contrasts = NULL, offset, ...)
## NULL
```

```r
y <- 10
x <- 2
  ## the lm() functions has many arguments, some of which
  ## have defaults
lm(y ~ x, mydata, 1:100, model = FALSE)
```

```
##
## Call:
## lm(formula = y ~ x, data = mydata, subset = 1:100, model = FALSE)
##
## Coefficients:
## (Intercept)            x
##          25            2
```

```r
lm(data = mydata, y ~ x, model = FALSE, 1:100)
```

```
##
## Call:
## lm(formula = y ~ x, data = mydata, subset = 1:100, model = FALSE)
##
## Coefficients:
## (Intercept)            x
##          25            2
```

```
  ## both commands give the same result. But mixing matching is
  ## not recommended.
```

Function arguments can also be partially matched for a named argument. When an argument is given, the program will, in order, attempt to match:
1. an exact name
2. a partial match
3. a positional match

In addition, you can also set an argument to *NULL*. It can mean a variety of things.

```
source("My_Functions.R")
```

```
## [1] "Code read"
```

```
args(above)
```

```
## function (x, n = 10)
## NULL
```

```
above(1:20, n = NULL)
```

```
## integer(0)
```

The arguments to a function are only evaluated as they are needed.

```
f <- function(a, b) {
        a^2
}
f(2)
```

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

```
        ## even though argument b is undefined and has no default,
        ## it still gives a logical value, because argument b is
        ## not used.

f <- function(a, b) {
        print(a)
        print(b)
}
f(2)
```

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

```
## Error in print(b): argument "b" is missing, with no default
```

```
        ## error only occurs when b is needed. the function
        ## works correctly up to then.
```

The "..." Argument

Indicates a variable number of arguments which are usually passed onto another function. Often used to extend another function, without copying the entire list of arguments from that function.

```
myplot <- function(x, y, type = "l", ...) {
            plot(x, y, type = type, ...)
}  ## imitates plot functions, but sets type to "l" by default.
```

Generic functions use ... so that extra arguments can be passed to methods (more on this later).

It is also necessary when the number of arguments cannot be known in advance.

```
args(paste)
```

```
## function (..., sep = " ", collapse = NULL, recycle0 = FALSE)
## NULL
```

```
x <- "Hello"
y <- "world!"
z <- "I'm"
w <- "R!"
paste(x, y, z, w, sep = " ", collapse = NULL, recycle0 = FALSE)
```

```
## [1] "Hello world! I'm R!"
```

```
      ## several arguments can be placed in the ... position.

args(cat)  ## similar function
```

```
## function (..., file = "", sep = " ", fill = FALSE, labels = NULL,
##     append = FALSE)
## NULL
```

You **CANNOT** use positional matching nor partial matching for arguments that come after the "..."

Scoping Rules

Symbol Binding

What happens if you assign a function string that already exists?

```
lm <- function(x) x * x
lm(5)
```

```
## [1] 25
```

the *lm()* function already exists and is defined in the stats package. But R will first search for the meaning of lm in the *global environment* (the workspace and objects set by the user). After that it works through a list of other environments in order, until it find it. The list can be viewed with the *search()* function.

```
search()
```

```
## [1] ".GlobalEnv"        "package:stats"     "package:graphics"
## [4] "package:grDevices" "package:utils"     "package:datasets"
## [7] "package:methods"   "Autoloads"         "package:base"
```

Order matters! Can be configured by the user. When loading a package with the *library()* function, that package gets put to position 2 in the list, after the global environment, and everything else is pushed down one level.
Note that R has separate namespaces for functions and non-functions. So it is possible to have an object named *c* and a function named *c*. (Though on the global environment there can only be one object named *c*)

R Scoping Rules

- Determine how a value is associated with a free variable in a function
- *lexical* aka *static* scoping. (an alternative is *dynamic* scoping)
- R uses the *search* list to bind a value to a symbol
- *lexical scoping* is particularly useful for sumplifying statistical computations.

```
f <- function(x, y) {
            x^2 + y / z
}
```

How does R assign a value to *z*, if it's not defined in the function header?

- *the values of free variables are searched for in the environment in which the function was defined.*
  – An environment is a collection of (symbol, value) pairs
  – every environment has a parent environment, and can have multiple children environments.
  – the only environment without a parent is the empty environment
  – a function + an environment = a closure (or function closure)
- First R searches for the free variable in the environment of the function. It then continues searching in the parent environment.
- the search continues down the sequence until it reaches the top-level environment.
- after that, it continues down the *search* list until it hits the empty environment. If no value is found, an error is shown.

```
make.power <- function(n) {
                pow <- function(x) {
                          x^n
                }
                pow
}  ## this function returns another function as its value.
   ## so i can use:
cube <- make.power(3)
square <- make.power(2)
   ## to make functions "cube" and "square" that call "pow"
   ## with a value
cube(5)
```

```
## [1] 125
```

```
square(5)
```

```
## [1] 25
```

- *ls(environment())* can be used to call the environment of a function. (*ls()* lists objects.)
- *get()* returns the value of an object, and can be used to find the value of arguments of a function, e.g.:

```
ls(environment(cube))
```

```
## [1] "n"   "pow"
```

```
get("n", environment((cube)))
```

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

```
ls(environment(square))
```

```
## [1] "n"   "pow"
```

```
get("n", environment((square)))
```

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

What does this mean?

```
y <- 10
f <- function(x) {
          y <- 2
          y^2 + g(x)
}

g <- function(x) {
          y * x
}
f(3)
```

```
## [1] 34
```

```
        ## results in 34 because y is set to 2 in the function f,
        ## but not in the function g
g(3)
```

```
## [1] 30
```

```
        ## results in 30 because the environment of g doesn't
        ## include setting y to 2.
```

- in *lexical* scoping the value of y is looked up in the environment where the function is defined.
- in *dynamic* scoping the value of y is looked up in the environment where the function is **called** (known as "calling environment")
  – in R this is called the "parent frame"

Consequences of Lexical Scoping

- In R all objects must be stored in memory.
- functions must carry a pointer to their respective environments, which could be anywhere.
- In S-PLUS, for comparison, free variables are always looked up in the global workspace. So everything can be stored in the disk, because the "defining environment" of all functions was the same.

Coding Standards

Making code readable.

- write your code using a text editor and save it as a text file.
  – these can be read by any editing program.
- indent your code. (8 spaces per tab?)
- limit the width of your code.(80 columns?).
  – forces you to limit nesting loops, which is not very readable.
- limit the length of individual functions: one basic activity.
  – helps you debug functions and which function is malfunctioning.

Dates and Times in R

Represented using special object classes:

- Dates are represented by the *Date* class
  – Dates are stored internally as the number of days since 1970-01-01

- Times are represented by the *POSIXct* or the *POSIXlt* class
  – Times are stored internally as the number of seconds since 1970-01-01
  – *POSIXct* is stored internally as a large integer, and is useful for storing times in something like a data frame
  – *POSIXlt* is stored internally as a list, and includes other information such as day of the week, day of the year, day of the month.
  – Functions that work on dates and times:
  — *weekdays*
  — *months*
  — *quarters*

```
x <- as.Date("1970-01-01")
x
```

```
## [1] "1970-01-01"
```

```
unclass(x)
```

```
## [1] 0
```

```
unclass(as.Date("1970-01-02"))
```

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

```
unclass(as.Date("1969-12-31"))
```

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

```
y <- Sys.time()
y
```

```
## [1] "2020-10-13 16:15:38 NZDT"
```

```
p <- as.POSIXlt(y)
names(unclass(p))
```

```
##  [1] "sec"    "min"    "hour"   "mday"   "mon"    "year"   "wday"   "yday"
##  [9] "isdst"  "zone"   "gmtoff"
```

```
p$sec
```

```
## [1] 38.72353
```

```
p$yday
```

```
## [1] 286
```

```
y  ## Sys.time() is already in POSIXct format
```

```
## [1] "2020-10-13 16:15:38 NZDT"
```

```
unclass(x)
```

```
## [1] 0
```

```
x$sec  ## returns error since it is stored as only an integer
```

```
## Error in x$sec: $ operator is invalid for atomic vectors
```

The *strptime* function helps convert dates and times written in different formats.

```
datestring <- c("January 10, 2020 16:45", "September 26, 1986 03:21")
x <- strptime(datestring, "%B %d, %Y %H:%M")
x
```

```
## [1] "2020-01-10 16:45:00 NZDT" "1986-09-26 03:21:00 NZST"
```

```
class(x)
```

```
## [1] "POSIXlt" "POSIXt"
```

```
## use the help page to remember what the strings are
```

You can use data and time objects to perform calculations, comparisons, etc. But they must be the same class. The operators keep track of tricky things like leap years and time-zones. Plotting functions also recognise date & time objects and interpret them in unique ways.

Week 2 Quiz Questions

```
f <- function(x) {
        g <- function(y) {
                y + z
        }
        z <- 4
        x + g(x)
}
z <- 10
f(3)
```

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

Week 2 Programming Assignment

Part 1

Write a function named 'pollutantmean' that calculates the mean of a pollutant (sulfate or nitrate) across a specified list of monitors. The function 'pollutantmean' takes three arguments: 'directory', 'pollutant', and 'id'. Given a vector monitor ID numbers, 'pollutantmean' reads that monitors' particulate matter data from the directory specified in the 'directory' argument and returns the mean of the pollutant across all of the monitors, ignoring any missing values coded as NA. A prototype of the function is as follows:

```
##  pollutantmean <- function(directory, pollutant, id = 1:332) {
                ## 'directory' is a character vector of length 1 indicating
                ## the location of the CSV file.

                ## 'pollutant' is a character value of length 1 indicating
                ## the name of the pollutant for which we will calculate the
                ## mean; either "sulfate" or "nitrate".

                ## 'id' is an integer vector indicating the monitor ID
                ## numbers to be used

                ## Return the mean of the pollutant across all monitors list
                ## in the 'id' vector (ignoring NA values_
                ## NOTE: Do not round the result)
##  }
```

You can see some example output from this function below. The function that you write should be able to match this output. Please save your code to a file named pollutantmean.R.

Sample results

```
source("pollutantmean.R")
a <- pollutantmean("specdata", "sulphate", 1:10)
b <- pollutantmean("specdata", "nitrate", 70:72)
c <- pollutantmean("specdata", "nitrate", 23)
c(a, b, c)
```

```
## [1] 4.064128 1.706047 1.280833
```

Part 2

Write a function that reads a directory full of files and reports the number of completely observed cases in each data file. The function should return a data frame where the first column is the name of the file and the second column is the number of complete cases. A prototype of this function follows:

```
complete <- function(directory, id = 1:322) {
        ## 'directory' is a character vector of length 1 indicating
        ## the location of the CSV files

        ## 'id' is an integer vector indicating the monitor ID numbers
        ## to be used
```

```
        ## Return a data frame of the form:
        ## id nobs
        ## 1 117
        ## 2 1041
        ## ...
        ## where 'id' is the monitor ID number and 'nobs' is the
        ## number of complete cases.
}
```

You can see some example output from this function below. The function that you write should be able to match this output. Please save your code to a file named complete.R. To run the submit script for this part, make sure your working directory has the file complete.R in it.

Sample results

```
source("complete.R")
complete("specdata", 1)
```

```
##   id nobs
## 1  1  117
```

```
complete("specdata", c(2, 4, 8, 10, 12))
```

```
##   id nobs
## 1  2 1041
## 2  4  474
## 3  8  192
## 4 10  148
## 5 12   96
```

```
complete("specdata", 30:25)
```

```
##   id nobs
## 1 30  932
## 2 29  711
## 3 28  475
## 4 27  338
## 5 26  586
## 6 25  463
```

```
complete("specdata", 3)
```

```
##   id nobs
## 1  3  243
```

Part 3

Write a function that takes a directory of data files and a threshold for complete cases and calculates the correlation between sulfate and nitrate for monitor locations where the number of completely observed cases (on all variables) is greater than the threshold. The function should return a vector of correlations for the monitors that meet the threshold requirement. If no monitors meet the threshold requirement, then the function should return a numeric vector of length 0. A prototype of this function follows:

```
corr <- function(directory, threshold = 0) {
## 'directory' is a character vector of length 1 indicating the location of
## the CSV files

## 'threshold' is a numeric vector of length 1 indicating the number of
## completely observed observations (on all variables) required to compute
## the correlation between nitrate and sulfate; the default is 0

## return a numeric vector of correlations
## NOTE: Do not round the result!
}
```

For this function you will need to use the 'cor' function in R which calculates the correlation between two vectors. Please read the help page for this function via '?cor' and make sure that you know how to use it.

You can see some example output from this function below. The function that you write should be able to approximately match this output. Note that because of how R rounds and presents floating point numbers, the output you generate may differ slightly from the example output. Please save your code to a file named corr.R. To run the submit script for this part, make sure your working directory has the file corr.R in it.

Sample Results

```
source("corr.R")
source("complete.R")
cr <- corr("specdata", 150)
head(cr)
```

```
## [1] -0.01895754 -0.14051254 -0.04389737 -0.06815956 -0.12350667 -0.07588814
```

```
summary(cr)
```

```
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -0.21057 -0.05147  0.09333  0.12401  0.26836  0.76313
```

```
cr <- corr("specdata", 400)
head(cr)
```

```
## [1] -0.01895754 -0.04389737 -0.06815956 -0.07588814  0.76312884 -0.15782860
```

```
summary(cr)
```

```
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -0.17623 -0.03109  0.10021  0.13969  0.26849  0.76313
```

```
cr <- corr("specdata", 5000)
summary(cr)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##
```

```
length(cr)
```

```
## [1] 0
```

```
cr <- corr("specdata")
summary(cr)
```

```
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -1.00000 -0.05282  0.10718  0.13684  0.27831  1.00000
```

```
length(cr)
```

```
## [1] 323
```

Swirl Exercises Week 2

Swirl Lesson 8: Logic

- Logical operators ==, *!=*, >, >=, <, <=
  - %% is used to find the remainder.
- *"!"* in front of a *TRUE* returns a *FALSE* and vice-versa

```
15 %% 7
```

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

```
TRUE == TRUE
```

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

```
(FALSE == TRUE) == FALSE
```

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

```
5 != 7
```

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

```
! 5 == 7
```

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

```
!(0 >= -1)
```

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

```
(TRUE != FALSE) == !(6 == 7)
```

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

- *"&"* and *"&&"* return TRUE only if both parts are TRUE
  – *"&"* evaluates each element in a vector (recycling may happen).
  – *"&&"* evaluates only the first element of vectors.

- *"|"* and *"||"* return TRUE if either parts are TRUE
  – same differences as for *"&"* and *"&&"*

- All **AND** operators are evaluated before **OR** operators.

```
TRUE & TRUE
```

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

```
FALSE & TRUE
```

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

```
FALSE & FALSE
```

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

```
TRUE & c(TRUE, FALSE, FALSE)
```

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

```
TRUE && c(TRUE, FALSE, FALSE)
```

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

```
6 != 10 && FALSE && 1 >= 2
```

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

```
5 > 8 || 6 != 8 && 4 > 3.9
```

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

- *isTRUE()* returns TRUE only if the argument inside is *TRUE*, otherwise it returns *FALSE*.

- *identical()* returns TRUE is both arguments are identical.

- *xor()* is the exclusive **OR**. It returns TRUE if only one element is TRUE.

- *which()* takes a logical vector as argument and returns the indeces of TRUE elements.

- *any()* and *all()* return TRUE if any or all elements of a logical vector are TRUE, respectively.

```
isTRUE("egg salad")
```

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

```
isTRUE(5 != 4)
```

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

```
identical("hello", "Hello")
```

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

```
xor(5 == 6, !FALSE)
```

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

```
xor(5 <= 6, 4*3 == 12)
```

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

```
which(c(TRUE, FALSE, TRUE))
```

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

```
any(c(1, 2, 3, 4, 5, 6) < 0)
```

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

```
all(c(1, 2, 3, 4, 5, 6) > 0)
```

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

Swirl Lesson 9: Functions

Functions as arguments

You can pass functions as arguments to other functions just like you can pass data to functions. Let's say you define the following functions:

```
add_two_numbers <- function(num1, num2){
    num1 + num2
}

multiply_two_numbers <- function(num1, num2){
  num1 * num2
}
```

```
some_function <- function(func){
    func(2, 4)
}

some_function(add_two_numbers)
```

```
## [1] 6
```

```
some_function(multiply_two_numbers)
```

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

```
some_function(paste)
```

```
## [1] "2 4"
```

As you can see we use the argument name "func" like a function inside of "some_function()."

Finish the function definition below so that if a function is passed into the "func" argument and some data (like a vector) is passed into the dat argument the evaluate() function will return the result of dat being passed as an argument to func.

```
evaluate <- function(func, dat){
  func(dat)
}

evaluate(sum, c(2, 4, 6))
```

```
## [1] 12
```

```
evaluate(median, c(7, 40, 9))
```

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

```
evaluate(floor, 11.1)
```

```
## [1] 11
```

Ellipses as arguments

The ellipses can be used to pass on arguments to other functions that are used within the function you're writing. Usually a function that has the ellipses as an argument has the ellipses as the last argument. The usage of such a function would look like: *ellipses_func(arg1, arg2 = TRUE, . . .)*
In the above example arg1 has no default value, so a value must be provided for arg1. arg2 has a default value, and other arguments can come after arg2 depending on how they're defined in the ellipses_func() documentation. Interestingly the usage for the paste function is as follows:
paste (. . . , sep = " ", collapse = NULL)
Notice that the ellipses is the first argument, and all other arguments after the ellipses have default values. This is a strict rule in R programming: all arguments after an ellipses must have default values. Take a look at the simon_says function below:

```r
simon_says <- function(...){
   paste("Simon says:", ...)
}
simon_says("put your hands on your head!")
```

```
## [1] "Simon says: put your hands on your head!"
```

Telegrams used to be peppered with the words START and STOP in order to demarcate the beginning and end of sentences. Write a function below called telegram that formats sentences for telegrams. For example the expression `telegram("Good", "morning")` should evaluate to: *"START Good morning STOP"*

```r
telegram <- function(...){
  paste("START", ..., "STOP")
}
telegram("Good", "morning")
```

```
## [1] "START Good morning STOP"
```

Unpacking Elipses

Let's explore how to "unpack" arguments from an ellipses when you use the ellipses as an argument in a function. Below I have an example function that is supposed to add two explicitly named arguments called alpha and beta.

```r
add_alpha_and_beta <- function(...){
   # First we must capture the ellipsis inside of a list
   # and then assign the list to a variable. Let's name this
   # variable `args`.

   args <- list(...)

   # We're now going to assume that there are two named arguments within args
   # with the names `alpha` and `beta.` We can extract named arguments from
   # the args list by using the name of the argument and double brackets. The
   # `args` variable is just a regular list after all!

   alpha <- args[["alpha"]]
   beta  <- args[["beta"]]

   # Then we return the sum of alpha and beta.

   alpha + beta
}

add_alpha_and_beta(alpha = 100, beta = 20)
```

```
## [1] 120
```

Have you ever played Mad Libs before? The function below will construct a sentence from parts of speech that you provide as arguments. We'll write most of the function, but you'll need to unpack the appropriate arguments from the ellipses.

```
mad_libs <- function(...){
  # Do your argument unpacking here!

  args <- list(...)
  place <- args["place"]
  adjective <- args["adjective"]
  noun <- args["noun"]

  # Don't modify any code below this comment.
  # Notice the variables you'll need to create in order for the code below to
  # be functional!
  paste("News from", place, "today where", adjective, "students took to the streets in protest of the ne
}
mad_libs(place = "Christchurch", adjective = "uncool", noun = "Ski ramp")
```

```
## [1] "News from Christchurch today where uncool students took to the streets in protest of the new Ski
```

Creating Binary Operators

The syntax for creating new binary operators in R is unlike anything else in R, but it allows you to define a new syntax for your function. I would only recommend making your own binary operator if you plan on using it often!
User-defined binary operators have the following syntax:
%[whatever]%
where [whatever] represents any valid variable name.
Let's say I wanted to define a binary operator that multiplied two numbers and then added one to the product. An implementation of that operator is below:

```
"%mult_add_one%" <- function(left, right){ # Notice the quotation marks!
   left * right + 1
}
4 %mult_add_one% 5
```

```
## [1] 21
```

Write your own binary operator below from absolute scratch! Your binary operator must be called %p% so that the expression:
"Good" %p% "job!"
will evaluate to: "Good job!"

```
"%p%" <- function(left, right){
  paste(left, right)
}
"Hello" %p% "world!"
```

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

Swirl Lesson 14: Dates and Times

- *Sys.Date()* gets a date object with the current date

- *as.Date("1986-09-26")* to create a custom date object
  – use *unclass(Sys.Date())* to see that dates are stored as days since 1970-01-01.
  – dates before 1970 are stored as negative numbers.

- *Sys.time()* is a POSIXct object with the current time.
  – time is stored internally as the number of seconds since 1970-01-01 00:00:00

- *as.POSIXlt(Sys.time())* coerces it into a POSIXlt object
  – it is stored internally as a list, with elements representing seconds, minutes, hours, day of month, month, year, day of week, day of year, daylight savings, time zone, and two more time zone related elements.

```
t1 <- as.POSIXlt("1986-09-26 12:45:00 BRT")
str(unclass(t1))  ## str() Compactly displays structure of an object.
```

```
## List of 11
##  $ sec   : num 0
##  $ min   : int 45
##  $ hour  : int 12
##  $ mday  : int 26
##  $ mon   : int 8
##  $ year  : int 86
##  $ wday  : int 5
##  $ yday  : int 268
##  $ isdst : int 0
##  $ zone  : chr "NZST"
##  $ gmtoff: int NA
```

```
t1$min  ## easily extracted elements
```

```
## [1] 45
```

- *weekdays()* returns day of the week for any date or time object.

- *months()* returns the month.

- *quarters()* returns the quarter of th year.
  – there functions work on date, POSIXct and POSIXlt objects.

```
t1 <- as.POSIXlt("1986-09-26 12:45:00 BRT")
weekdays(t1)
```

```
## [1] "Friday"
```

```
months(t1)
```

```
## [1] "September"
```

```
quarters(t1)
```

```
## [1] "Q3"
```

- *strptime()* can extract dates and times from formats not recognised by R.
  – it converts a character vector into POSIXlt.

```
t1 <- "October 17, 1986 08:24"
strptime(t1, "%B %d, %Y %H:%M")
```

```
## [1] "1986-10-17 08:24:00 NZST"
```

- arithmetic operations + and -, and comparisons <, > and == can be performed on date and time objects.

- *difftime()* allows for more precise control of units when performing subtractions.