Structure*

Chapter 3

Anna Ziff

R Workflow for Economists

Contents

*Please contact anna.ziff@duke.edu if there are errors.

Logical Operations

Recall from chapter 1 that a logical object can take two values: `TRUE` or `FALSE`. Logical operators, those that have logical objects as inputs or outputs, are commonly used and crucial to understand before we discuss conditional statements and loops. We will define a, b, c, and d to help demonstrate logical operators.

```
a <- 3
b <- 8
c <- c(1, 9, 3)
d <- c(7, 10, 3)
```

The inequalities, $<, >, \leq, \geq$, are denoted `<, >, <=, >=`, respectively.

```
a > b
```

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

```
a <= b
```

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

Note that we can also apply these operators to vectors with more than one element.

```
c
```

```
## [1] 1 9 3
```

```
d
```

```
## [1]  7 10  3
```

```
c < d
```

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

The operator `==` allows us to test if two objects are equal. The operator `!=` allows us to test if two objects are not equal.

```
a == b
```

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

```
c != d
```

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

Generally, `!` indicates negation.

```
is.character(b)
```

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

```
!is.character(b)
```

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

We can combine logical operators using `&` (conjunction, “and”) or `|` (disjunction, “or”).

```
(a > b) & (b != 7)
```

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

```
(a > b) | (b != 7)
```

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

If there are many logical comparisons, it is useful to have functions that output `TRUE` if any or all of the comparisons are true.

```
c < d
```

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

```
any(c < d)
```

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

```
all(c < d)
```

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

If you are comparing objects with more than one element, you may want `TRUE` if the two objects are identical and `FALSE` otherwise. That is, you do not want a logical vector with more than one element.

```
c == d
```

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

```
identical(c, d)
```

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

Practice Exercises 3.1

1. Define a vector `x` of 10 random draws from the uniform $[0, 1]$ distribution. Test if each element is greater than or equal to 0.9.
2. Define a variable that is `TRUE` if at least 1 element of `x` is greater than or equal to 0.9.

Conditional Statements

Conditional statements allow you to dictate different actions depending on the outcome of a condition. The general syntax is as follows.

```
if (<condition>) {
  <command if condition is true>
}
```

If the condition, denoted in a general way as `<condition>`, is true, then the commands inside the brackets are executed. Otherwise, if the condition is false, R will continue executing the next line of code. Here is a concrete example.

```
f <- runif(1, min = 0, max = 1)
f
```

```
## [1] 0.2128504
```

```
if (f < 0.5) {
  print("The random number is less than one half.")
}
```

```
## [1] "The random number is less than one half."
```

If you want R to execute a different command if the condition is false, then the general code is as follows.

```
if (<condition>) {
  <command if condition is true>
} else {
  <command if condition is false>
}
```

Here is a concrete example.

```
f
```

```
## [1] 0.2128504
```

```
if (f < 0.5) {
  print("The random number is less than one half.")
} else {
  print("The random number is greater than or equal to one half.")
}
```

```
## [1] "The random number is less than one half."
```

It is possible that there are more than two logical possibilities.

```
f
```

```
## [1] 0.2128504
```

```
if (f < 0.25) {
  print("The random number is less than one quarter.")
} else if (f >= 0.25 & f < 0.5) {
  print("The random number is between one quarter and one half.")
} else if (f >= 0.5 & f < 0.75) {
  print("The random number is between one half and three quarters.")
} else {
  print("The random number is between three quarters and one.")
}
```

```
## [1] "The random number is less than one quarter."
```

Conditional statements can be nested. It is always best practice to have consistent use of indentations and brackets, but these conventions are especially important when nesting conditional statements. The below chunk produces the same results as the above chunk, but using nested conditional statements rather than `else if`.

```
f
```

```
## [1] 0.2128504
```

```
if (f < 0.5) {
  if (f < 0.25) {
    print("The random number is less than one quarter.")
  }
  else {
    print("The random number is between one quarter and one half.")
  }
} else {
  if (f < 0.75) {
    print("The random number is between one half and three quarters.")
  } else {
    print("The random number is between three quarters and one.")
  }
}
```

```
## [1] "The random number is less than one quarter."
```

It is important to be careful with the conditions you use. A common error is to have a condition that has more than one element (i.e., a logical vector with length greater than 1). In this case, R will use the first element and throw a warning.

```
g <- runif(2, min = 0, max = 1)
g
```

```
## [1] 0.2964665 0.4137872
```

```
if (g < 0.5) {
  print("The random number is less than one half.")
}
```

```
## Warning in if (g < 0.5) {: the condition has length > 1 and only the first
## element will be used
```

```
## [1] "The random number is less than one half."
```

```
g < 0.5
```

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

The function `ifelse()` is sometimes useful when applying functions to vectors or assigning values to a variable. It follows the same logic as conditional statements, but allows for a more concise implementation.

```
x <- 2:-1
y <- ifelse(x >= 0, sqrt(x), NA)
```

```
## Warning in sqrt(x): NaNs produced
```

```
ifelse(is.na(y), 0, 1)
```

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

Practice Exercises 3.2

1. Run the following (incorrect) code. How can you change the condition to avoid the warning and ensure that the print out is correct?

```
g <- runif(2, min = 0, max = 1)
g
if (g < 0.5) {
  print("All the random numbers are less than one half.")
}
```

Loops

Loops indicate portions of the code to be executed more than once. The loop ends when the number of iterations has been reached or there is an exit condition.

`for`

The `for` instruction involves running the code a pre-specified number of iterations. The general syntax is as follows.

```
for (i in <vector>) {
  <commands>
}
```

R iterates through each value of `<vector>` and executes the commands inside the brackets. Once the last element of the vector is reached, R executes the next line of code. Here is an example.

```
for (i in 1:3) {
  print(factorial(i))
}
```

```
## [1] 1
## [1] 2
## [1] 6
```

`while`

The `while` instruction involves running the code until an exit condition is satisfied. The general syntax is as follows.

```
while (<condition>) {
  <command>
}
```

Here is an example.

```
j <- 1
while (j < 4) {
  print(factorial(j))
  j <- j + 1
}
```

```
## [1] 1
## [1] 2
## [1] 6
```

Other Loop Instructions

In some contexts, it is useful to further control a loop. The instruction `break` tells R to exit the loop.

```
l <- c(2, 4, 7)
for (i in l) {
  if (i == 4) {
    out <- i
    break
  }
}
out
```

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

The instruction `next` tells R to move to the next iteration.

```
for (i in l) {
  if (i == 4) {
    next
  }
  print(i)
}
```

```
## [1] 2
## [1] 7
```

Efficiency

A common maxim that loops are slow in R and it is best to avoid them. It is true that there are faster alternatives. The use of vectorized operations is preferable when possible as these operations are incredibly fast. The function `system.time()` allows you to test the speed of your code. Evidently, the vectorized operation `factorial(1:100000)` is faster than the loop. The divergence in speeds between the two methods will be larger for more complex functions and more iterations.

```
system.time(for (i in 1:100000) {
  factorial(i)
})
```

```
##    user  system elapsed
##   0.018   0.002   0.020
```

```
system.time(factorial(1:100000))
```

```
##    user  system elapsed
##       0       0       0
```

Regardless, it is still crucial to be comfortable with loops. For smaller computations, like factorial, the difference is very minor. Sometimes, the code is clearer and makes more sense with a loop than with a vectorized operation.

Practice Exercises 3.3

1. Define an object `out` with the value 0. In a `for` loop iterating 1, 2, . . . , 20, add the reciprocal to `out` if the number is even. The sum should be $\frac{1}{2} + \frac{1}{4} + \ldots + \frac{1}{20}$.

Apply Functions

A family of functions allows you to take advantage of R's efficiency with vectorized operations, rather than relying too heavily on loops. There are several different functions within this family that differ by what object the function is applied and the desired output.

`apply()`

The function `apply()` applies a function to the rows or columns (these are called margins) of matrices or data frames. While it is not faster than loops, it allows for more compact code. Following chapter 19 in Boehmke (2016), we will use the built-in data frame `mtcars`. To see the first 6 rows of `mtcars`, use the `head()` function.

```
head(mtcars)
```

```
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

If we want the mean of each column, we can use the `apply()` function. Note that `2` corresponds to the second margin of the data frame, i.e., the columns. Notice that the output is a named vector.

```
x <- apply(mtcars, 2, mean)
x
```

```
##        mpg        cyl       disp         hp       drat         wt       qsec
```

```
## 20.090625   6.187500 230.721875 146.687500   3.596563   3.217250  17.848750
##         vs         am       gear       carb
##   0.437500   0.406250   3.687500   2.812500
```

```r
str(x)
```

```
##  Named num [1:11] 20.09 6.19 230.72 146.69 3.6 ...
##  - attr(*, "names")= chr [1:11] "mpg" "cyl" "disp" "hp" ...
```

The first margin of the data frame is row.

```r
apply(mtcars, 1, max)
```

```
##           Mazda RX4       Mazda RX4 Wag          Datsun 710      Hornet 4 Drive
##               160.0               160.0               108.0               258.0
##   Hornet Sportabout             Valiant          Duster 360           Merc 240D
##               360.0               225.0               360.0               146.7
##            Merc 230            Merc 280           Merc 280C          Merc 450SE
##               140.8               167.6               167.6               275.8
##          Merc 450SL         Merc 450SLC  Cadillac Fleetwood Lincoln Continental
##               275.8               275.8               472.0               460.0
##   Chrysler Imperial            Fiat 128         Honda Civic      Toyota Corolla
##               440.0                78.7                75.7                71.1
##       Toyota Corona    Dodge Challenger         AMC Javelin          Camaro Z28
##               120.1               318.0               304.0               350.0
##    Pontiac Firebird           Fiat X1-9       Porsche 914-2        Lotus Europa
##               400.0                79.0               120.3               113.0
##      Ford Pantera L        Ferrari Dino       Maserati Bora          Volvo 142E
##               351.0               175.0               335.0               121.0
```

If the function to be applied has other arguments, these can be specified separated by commas. Here is an example where we trim 10% of observations from each end.

```r
out <- apply(mtcars, 2, mean, trim = 0.1)
out
```

```
##         mpg         cyl        disp          hp        drat          wt
##  19.6961538   6.2307692 222.5230769 141.1923077   3.5792308   3.1526923
##        qsec          vs          am        gear        carb
##  17.8276923   0.4230769   0.3846154   3.6153846   2.6538462
```

There are some function that are faster than the analogous implementation in `apply()`. These include `summary()`, `colSums()`, `rowSums()`, `colMeans()`, and `rowMeans()`.

```r
colMeans(mtcars)
```

```
##        mpg        cyl       disp         hp       drat         wt       qsec
##  20.090625   6.187500 230.721875 146.687500   3.596563   3.217250  17.848750
##         vs         am       gear       carb
##   0.437500   0.406250   3.687500   2.812500
```

```r
summary(mtcars)
```

```
##       mpg             cyl             disp             hp
##  Min.   :10.40   Min.   :4.000   Min.   : 71.1   Min.   : 52.0
##  1st Qu.:15.43   1st Qu.:4.000   1st Qu.:120.8   1st Qu.: 96.5
##  Median :19.20   Median :6.000   Median :196.3   Median :123.0
##  Mean   :20.09   Mean   :6.188   Mean   :230.7   Mean   :146.7
##  3rd Qu.:22.80   3rd Qu.:8.000   3rd Qu.:326.0   3rd Qu.:180.0
```

```
##  Max.   :33.90   Max.   :8.000   Max.   :472.0   Max.   :335.0  
##       drat             wt             qsec             vs        
##  Min.   :2.760   Min.   :1.513   Min.   :14.50   Min.   :0.0000  
##  1st Qu.:3.080   1st Qu.:2.581   1st Qu.:16.89   1st Qu.:0.0000  
##  Median :3.695   Median :3.325   Median :17.71   Median :0.0000  
##  Mean   :3.597   Mean   :3.217   Mean   :17.85   Mean   :0.4375  
##  3rd Qu.:3.920   3rd Qu.:3.610   3rd Qu.:18.90   3rd Qu.:1.0000  
##  Max.   :4.930   Max.   :5.424   Max.   :22.90   Max.   :1.0000  
##        am              gear            carb      
##  Min.   :0.0000   Min.   :3.000   Min.   :1.000  
##  1st Qu.:0.0000   1st Qu.:3.000   1st Qu.:2.000  
##  Median :0.0000   Median :4.000   Median :2.000  
##  Mean   :0.4062   Mean   :3.688   Mean   :2.812  
##  3rd Qu.:1.0000   3rd Qu.:4.000   3rd Qu.:4.000  
##  Max.   :1.0000   Max.   :5.000   Max.   :8.000
```

```
summary(mtcars$mpg)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
##   10.40   15.43   19.20   20.09   22.80   33.90
```

lapply()

The `lapply()` function is designed to apply functions to lists and return a list. It efficiently loops through a list and applies the specified function to each element.

```
carlist <- as.list(mtcars[1:5, ]) # Convert the first 5 rows into a list
str(carlist)
```

```
## List of 11
##  $ mpg : num [1:5] 21 21 22.8 21.4 18.7
##  $ cyl : num [1:5] 6 6 4 6 8
##  $ disp: num [1:5] 160 160 108 258 360
##  $ hp  : num [1:5] 110 110 93 110 175
##  $ drat: num [1:5] 3.9 3.9 3.85 3.08 3.15
##  $ wt  : num [1:5] 2.62 2.88 2.32 3.21 3.44
##  $ qsec: num [1:5] 16.5 17 18.6 19.4 17
##  $ vs  : num [1:5] 0 0 1 1 0
##  $ am  : num [1:5] 1 1 1 0 0
##  $ gear: num [1:5] 4 4 4 3 3
##  $ carb: num [1:5] 4 4 1 1 2
```

```
lapply(carlist, mean)
```

```
## $mpg
## [1] 20.98
## 
## $cyl
## [1] 6
## 
## $disp
## [1] 209.2
## 
## $hp
## [1] 119.6
## 
## $drat
```

```
## [1] 3.576
##
## $wt
## [1] 2.894
##
## $qsec
## [1] 17.71
##
## $vs
## [1] 0.4
##
## $am
## [1] 0.6
##
## $gear
## [1] 3.6
##
## $carb
## [1] 2.4
```

The elements of the list `carlist` are all vectors. A list may contain matrices or data frames as well. In that case, we may want to iterate through each element of the list and apply the function to each item of each element. This is efficiently done through nesting apply functions. To demonstrate, we create a list in which each element is a data frame.

```
carlist <- list(mazda = mtcars[1:2, ], hornet = mtcars[4:5, ], merc = mtcars[8:14, ])
carlist
```

```
## $mazda
##               mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4      21   6  160 110  3.9 2.620 16.46  0  1    4    4
## Mazda RX4 Wag  21   6  160 110  3.9 2.875 17.02  0  1    4    4
##
## $hornet
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
##
## $merc
##             mpg cyl  disp  hp drat   wt qsec vs am gear carb
## Merc 240D   24.4   4 146.7  62 3.69 3.19 20.0  1  0    4    2
## Merc 230    22.8   4 140.8  95 3.92 3.15 22.9  1  0    4    2
## Merc 280    19.2   6 167.6 123 3.92 3.44 18.3  1  0    4    4
## Merc 280C   17.8   6 167.6 123 3.92 3.44 18.9  1  0    4    4
## Merc 450SE  16.4   8 275.8 180 3.07 4.07 17.4  0  0    3    3
## Merc 450SL  17.3   8 275.8 180 3.07 3.73 17.6  0  0    3    3
## Merc 450SLC 15.2   8 275.8 180 3.07 3.78 18.0  0  0    3    3
```

The `x` is a stand-in value.

```
lapply(carlist, function(x) apply(x, 2, mean))
```

```
## $mazda
##      mpg      cyl     disp       hp     drat       wt     qsec       vs
##  21.0000   6.0000 160.0000 110.0000   3.9000   2.7475  16.7400   0.0000
##       am     gear     carb
##   1.0000   4.0000   4.0000
```

```
##
## $hornet
##      mpg      cyl     disp       hp     drat       wt     qsec       vs
##  20.0500   7.0000 309.0000 142.5000   3.1150   3.3275  18.2300   0.5000
##       am     gear     carb
##   0.0000   3.0000   1.5000
##
## $merc
##         mpg         cyl        disp          hp        drat          wt
##  19.0142857   6.2857143 207.1571429 134.7142857   3.5228571   3.5428571
##        qsec          vs          am        gear        carb
##  19.0142857   0.5714286   0.0000000   3.5714286   3.0000000
```

sapply()

The function `sapply()` is very similar to `lapply()` except it outputs a simplified result whenever possible. If the output is a list with elements of length 1 (more than 1), `sapply()` returns a vector (matrix). Otherwise, `sapply()` returns a list.

```r
sapply(carlist, function(x) apply(x, 2, mean))
```

```
##         mazda   hornet        merc
## mpg   21.0000  20.0500  19.0142857
## cyl    6.0000   7.0000   6.2857143
## disp 160.0000 309.0000 207.1571429
## hp   110.0000 142.5000 134.7142857
## drat   3.9000   3.1150   3.5228571
## wt     2.7475   3.3275   3.5428571
## qsec  16.7400  18.2300  19.0142857
## vs     0.0000   0.5000   0.5714286
## am     1.0000   0.0000   0.0000000
## gear   4.0000   3.0000   3.5714286
## carb   4.0000   1.5000   3.0000000
```

tapply()

The function `tapply()` efficiently applies functions over subsets of a vector. It is useful when you want to apply functions within groups. The below code calculates the average miles per gallon (`mpg`) grouped by the number of cylinders (`cyl`).

```r
tapply(mtcars$mpg, mtcars$cyl, mean)
```

```
##        4        6        8
## 26.66364 19.74286 15.10000
```

Here is an example of the same situation but for each column in the data frame.

```r
apply(mtcars, 2, function(x) tapply(x, mtcars$cyl, mean))
```

```
##        mpg cyl     disp        hp     drat       wt     qsec        vs
## 4 26.66364   4 105.1364  82.63636 4.070909 2.285727 19.13727 0.9090909
## 6 19.74286   6 183.3143 122.28571 3.585714 3.117143 17.97714 0.5714286
## 8 15.10000   8 353.1000 209.21429 3.229286 3.999214 16.77214 0.0000000
##          am     gear     carb
## 4 0.7272727 4.090909 1.545455
## 6 0.4285714 3.857143 3.428571
## 8 0.1428571 3.285714 3.500000
```

Practice Exercises 3.4

1. Create a list with 4 elements, each containing a vector with 30 numbers: the first element is 30 draws from the uniform $[0, 1]$ distribution, the second element is 30 draws from the uniform $[1, 2]$ distribution, the third element is 30 draws from the uniform $[2, 3]$ distribution, and the fourth element is 30 draws from the uniform $[3, 4]$ distribution. How does `lapply` and `sapply` differ here?
2. Here is an example of implementing the function `quantile`. Calculate the 25th, 50th, and 75th percentile for the columns of `mtcars` using `apply()`.

```
quantile(1:100, probs = c(0.10, 0.90))
```

```
##  10%  90%
## 10.9 90.1
```

Functions

By now, you have been exposed to functions, both those built into the base packages of R, and those that are accessible from external packages. Now, you will write your own functions. User-defined functions are crucial in the workflow. They allow for tasks to be more general and automatic. Often, they make the script easier to read and understand. They also make it easier to test and debug, resulting in more correct output. Even if it feels burdensome to write functions, it is usually worthwhile.

The example function we will be using is very simple from Boehmke (2016). Suppose we want a function to calculate the present value of a future value given interest rate and number of periods. The output should be rounded to three digits.

```
calc_pv <- function(fv, r, n) {
  pv <- fv / ((1 + r)^n)
  return(round(pv, 3))
}
```

There are three components of the function. The `body` is the meat of the function. It contains all the calculations, that is, everything between `{}`.

```
body(calc_pv)
```

```
## {
##     pv <- fv/((1 + r)^n)
##     return(round(pv, 3))
## }
```

The `formals` are the inputs the function requires. These are also called arguments.

```
formals(calc_pv)
```

```
## $fv
##
##
## $r
##
##
## $n
```

Finally, the `environment` includes all of the named objects accessible to the function.

```
environment(calc_pv)
```

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

The function is called the same way as built-in functions.

```
calc_pv(fv = 1000, r = 0.08, n = 10)
```

```
## [1] 463.193
```

```
calc_pv(1000, 0.08, 10) # Positional matching
```

```
## [1] 463.193
```

It might be convenient to set default values for some arguments. You can see examples of default values in many built-in functions. In the documentation for `mean()`, the arguments `trim` and `na.rm` are listed in the description with `trim = 0` and `na.rm = FALSE`. Setting default values for your own code can help prevent errors and make the function easier to use.

```
calc_pv <- function(fv, r = 0.08, n) {
  pv <- fv / ((1 + r)^n)
  return(round(pv, 3))
}
```

Now, you can use the function even without specifying `r`.

```
calc_pv(fv = 1000, n = 20)
```

```
## [1] 214.548
```

If you specify a function with an argument that is not used, it will not throw an error or warning. The technical name for this is lazy evaluation.

```
calc_pv <- function(fv, r, n, x) {
  pv <- fv / ((1 + r)^n)
  return(round(pv, 3))
}
calc_pv(fv = 1000, r = 0.08, n = 10) # No need to pass a value to x
```

```
## [1] 463.193
```

In the present value example, the output is one number. Functions can also return more than one object. Above, we used the function `return()` to be very clear about what the function is returning. While this is good practice, the function will default to returning the last object of the body.

```
arith <- function(x, y) {
  x + y
  x - y
  x * y
  x / y
}
arith(1, 2)
```

```
## [1] 0.5
```

Returning a vector allows the function to return more than one result.

```
arith <- function(x, y) {
  addition <- x + y
  subtraction <- x - y
  multiplication <- x * y
  division <- x / y
  c(addition, subtraction, multiplication, division)
}
arith(1, 2)
```

```
## [1]  3.0 -1.0  2.0  0.5
```

Returning a list allows the function to return more than one result *and* allows for an easier understanding of the output.

```
arith <- function(x, y) {
  addition <- x + y
  subtraction <- x - y
  multiplication <- x * y
  division <- x / y
  c(list(Addition = addition, Subtraction = subtraction,
         Multiplication = multiplication, Division = division))
}
arith(1, 2)
```

```
## $Addition
## [1] 3
##
## $Subtraction
## [1] -1
##
## $Multiplication
## [1] 2
##
## $Division
## [1] 0.5
```

Scoping

When writing functions, it is useful to have a sense of the scoping rules. These are the rules that R follows to decide on the value of the objects in a function. R will first search within the function. If all the objects are defined within the function, R does not search anymore.

```
calc_pv <- function() {
  fv <- 1000
  r <- 0.08
  n <- 10
  fv / ((1 + r)^n)
}
calc_pv()
```

```
## [1] 463.1935
```

If a value is not present within the function, R will expand the search up one level. The levels are demarcated with `{}`.

```
fv <- 1000
calc_pv <- function() {
  r <- 0.08
  n <- 10
  fv / ((1 + r)^n)
}
calc_pv()
```

```
## [1] 463.1935
```

These rules are general, including when the function takes arguments.

```
calc_pv <- function(fv, r) {
  n <- 10
  fv / ((1 + r)^n)
}
calc_pv(1000, 0.08)
```

```
## [1] 463.1935
```

Niceties

Our example functions are simple, but often actual functions can be complex. Your script may define and use many functions that have many inputs and outputs. Even if you are the only person who will ever read your code, it is useful to include some checks to help ensure you are properly using your function. Even though these checks take time, they can save trouble down the line. The function `stop()` stops the execution of the function and throws an error message.

```
calc_pv <- function(fv, r, n) {

  # Input validation
  if (!is.numeric(fv) | !is.numeric(r) | !is.numeric(n)) {
    stop("All inputs must be numeric.\n",
         "The inputs are of the following classes:\n",
         "fv: ", class(fv), "\n",
         "r: ", class(r), "\n",
         "n: ", class(n))
  }

  pv <- fv / ((1 + r)^n)
  return(round(pv, 3))

}
calc_pv("1000", 0.08, 10)
```

```
## Error in calc_pv("1000", 0.08, 10): All inputs must be numeric.
## The inputs are of the following classes:
## fv: character
## r: numeric
## n: numeric
```

Notice that our function can take in vectors. As discussed above, vectorized operations like this are very efficient in R.

```
calc_pv(fv = 1:10, r = 0.08, n = 10)
```

```
##  [1] 0.463 0.926 1.390 1.853 2.316 2.779 3.242 3.706 4.169 4.632
```

But what if one of the elements of the input vector is a missing value?

```
calc_pv(fv = c(100, NA, 1000), r = 0.08, n = 10)
```

```
## [1]  46.319      NA 463.193
```

This is a frequent issue that arises when using functions within a larger script. Adding the argument `na.rm` allows you to specify how you want the function to handle `NA` values.

```
calc_pv <- function(fv, r, n, na.rm = FALSE) {

  # Input validation
```

```r
  if (!is.numeric(fv) | !is.numeric(r) | !is.numeric(n)) {
    stop("All inputs must be numeric.\n",
         "The inputs are of the following classes:\n",
         "fv: ", class(fv), "\n",
         "r: ", class(r), "\n",
         "n: ", class(n))
  }

  # na.rm argument
  if (na.rm == TRUE) {
    fv <- fv[!is.na(fv)] # Only keep non-missing values in fv
  }

  pv <- fv / ((1 + r)^n)
  return(round(pv, 3))

}
calc_pv(fv = c(100, NA, 1000), r = 0.08, n = 10)
```

```
## [1]  46.319      NA 463.193
```

```r
calc_pv(fv = c(100, NA, 1000), r = 0.08, n = 10, na.rm = TRUE)
```

```
## [1]  46.319 463.193
```

With input validation and conditional statements, the code to write functions can be very long. Just like in Matlab, you can write your function in a separate script. The script should contain the entirety of the code for the function.

```r
# Title: calc_pv.R

calc_pv <- function(fv, r, n, na.rm = FALSE) {

  # Input validation
  if (!is.numeric(fv) | !is.numeric(r) | !is.numeric(n)) {
    stop("All inputs must be numeric.\n",
         "The inputs are of the following classes:\n",
         "fv: ", class(fv), "\n",
         "r: ", class(r), "\n",
         "n: ", class(n))
  }

  # na.rm argument
  if (na.rm == TRUE) {
    fv <- fv[!is.na(fv)] # Only keep non-missing values in fv
  }

  pv <- fv / ((1 + r)^n)
  return(round(pv, 3))

}
```

The function `source()` allows you to add the function to your global environment.

```r
# Title: main.R
```

```r
source("calc_pv.R")
calc_pv(fv = c(100, NA, 1000), r = 0.08, n = 10, na.rm = TRUE)
```

Practice Exercises 3.5

1. Write a function to convert fahrenheit to celsius. Define a 10×10 matrix with 100 draws from the uniform $[-10, 100]$ distribution. Test your function on this matrix.
2. The Fibonacci sequence is recursive: $x_n = x_{n-1} + x_{n-2}$. Write a function that takes n as an argument and computes x_n. Recall that $x_0 = 0$ and $x_1 = 1$. Use a `for` loop to print the first 15 elements of the Fibonacci sequence. Bonus: try using `sapply` to print the first 15 elements of the Fibonacci sequence.

Further Reading

The information from this chapter comes from chapters 18, 19 of Boehmke (2016) and chapters 5.2, 5.7-5.8 of Zamora Saiz et al. (2020).

References

Boehmke, Bradley C. 2016. *Data Wrangling with R*. Use R! Springer. https://link-springer-com.proxy.lib.duke.edu/content/pdf/10.1007%2F978-3-319-45599-0.pdf.

Zamora Saiz, Alfonso, Carlos Quesada González, Lluís Hurtado Gil, and Diego Mondéjar Ruiz. 2020. *An Introduction to Data Analysis in R: Hands-on Coding, Data Mining, Visualization and Statistics from Scratch*.