# Introduction to Programming with R - June 15

*Eric Archer (eric.archer@noaa.gov, 858-546-7121)*

## Taking a random sample from a vector

The `sample` function is a very useful function for drawing random samples from vectors either with or without replacement.

If it is called with just a vector, it returns a permuted form of that vector:

```
sample(1:10)  Size not specified, defaults to range from 1st argument
```

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

A smaller vector can be created by specifying the `size` argument:

```
sample(1:10, size = 5)
```

```
[1] 10  9  5  4  8
```

In this case, 5 unique values are returned. If you want to sample with replacement, specify `replace = TRUE`:

```
sample(1:10, size = 5, replace = TRUE)
```

```
[1]  8  7  7  8 10
```

If a larger vector is to be sampled, then `replace` has to be `TRUE`:

```
sample(1:10, size = 100, replace = TRUE)
```

```
  [1] 10  1  9  6  2  7  1  6  8  7  3  3  4  4  2  8  9 10 10  8  8  4  7
 [24]  8  2  4  1 10  5  2  6  5  1  4 10  6  4  1  9  5  5  9 10  9  2  8
 [47]  5  7  3  4  4  3  5  6  4  6  6  7  9  9  8  2  9  2  6  3  6  2  6
 [70]  4  8  9 10  1  7  4  1  5 10  2  9  9  2  8  1  2  8  4  7  8  5  9
 [93]  1  5  7  9  2  7 10  1
```

By default, all elements in the vector have the same likelihood of being sampled. Weights can be applied by specifying them in the `prob` argument:

```
sample(letters[1:5], size = 20, replace = TRUE, prob = c(10, 10, 1, 1, 0))
```

```
 [1] "b" "a" "b" "a" "a" "a" "b" "a" "b" "b" "b" "a" "a" "a" "b" "a" "a"
[18] "a" "a" "b"
```

## Discrete values

The function `unique()` will list the unique values in a vector in the order it finds them:

```
x <- sample(letters, 10, replace = TRUE)
x
```

```
 [1] "t" "d" "x" "f" "e" "s" "c" "e" "i" "e"
```

```
unique(x)
```

```
[1] "t" "d" "x" "f" "e" "s" "c" "i"
```

The function `duplicated()` will identify those elements in a vector that occur at an earlier position:

```r
duplicated(x)
```

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

```r
# the negation of duplicated is the same as unique
x[!duplicated(x)]
```

```
[1] "t" "d" "x" "f" "e" "s" "c" "i"
```

```r
unique(x)
```

```
[1] "t" "d" "x" "f" "e" "s" "c" "i"
```

To calculate the frequency of values in a vector (the number of occurences), use `table()`:

```r
x <- sample(letters, 20, replace = TRUE)
table(x)
```

```
x
b c d e f l n o p q r w x
1 1 3 2 1 1 2 2 1 1 2 2 1
```

`table` can be used for cross-tabulation as well - counting frequency of occurence of a combination of categories

```r
months <- sample(month.abb, 20, replace = TRUE)
sex <- sample(c("m", "f"), 20, replace = TRUE)
freq <- table(sex, months)
freq
```

```
   months
sex Apr Aug Feb Jan Jun Mar May Nov Oct Sep
  f   0   3   1   1   1   3   0   3   1   3
  m   1   0   0   0   0   0   1   1   1   0
```

The values in a table can be accessed like a vector or matrix

```r
freq["m", ]
```

```
Apr Aug Feb Jan Jun Mar May Nov Oct Sep
  1   0   0   0   0   0   1   1   1   0
```

```r
freq["f", c("Jan", "Feb", "Mar")]
```

```
Jan Feb Mar
  1   1   3
```

## Data selection and manipulation

To identify values of one vector that are within another one, use `%in%`:

```r
letters %in% c("a", "f", "G", "b")
```

```
 [1]  TRUE  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
[12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[23] FALSE FALSE FALSE FALSE
```

To identify values of a logical vector that are `TRUE`, use `which`:

```r
x <- sample(1:100, 20)
x
```

```
 [1]  64  90 100  66  35  36  22  52  78  63  83  58  25  93  20  17  68
[18]  79  67  41
```

```
which(x < 50)
```

```
[1]  5  6  7 13 15 16 20
```

To identify the minimum and maximum values, use `which.min` and `which.max`:

```
which.min(x)
```

```
[1] 16
```

```
which.max(x)
```

```
[1] 3
```

To see if any values meet a condition, use `any`: (logical)

```
any(x < 50)
```

```
[1] TRUE
```

```
any(x > 200)
```

```
[1] FALSE
```

To see if all values meet a condition, use `all`:

```
all(x < 50)
```

```
[1] FALSE
```

```
all(x < 200)
```

```
[1] TRUE
```

Vectors can be reversed with `rev`:

```
x <- sample(1:5, 10, replace = T)
x
```

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

```
rev(x)
```

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

and sorted with `sort`:

```
sort(x)
```

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

```
# in decreasing order
sort(x, decreasing = TRUE)
```

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

However, sort can't be applied to a matrix or data.frame to sort the rows. For that, you need `order`. `order` returns a vector of indices in the order they should be as if they were sorted:

```
x <- data.frame(
  v1 = sample(letters, 20, replace = TRUE),
  v2 = sample(letters, 20, replace = TRUE),
  v3 = sample(letters, 20, replace = TRUE)
```

```
)
x
```

```
   v1 v2 v3
1   k  g  i
2   s  b  v
3   k  a  k
4   s  r  y
5   g  v  e
6   h  b  x
7   q  o  b
8   b  f  y
9   q  z  s
10  g  u  r
11  h  x  w
12  t  u  u
13  s  h  h
14  u  p  n
15  n  w  q
16  r  b  x
17  d  n  z
18  r  r  x
19  a  d  b
20  f  k  f
```

```
x.ord <- order(x$v1)
x[x.ord, ]
```

```
   v1 v2 v3
19  a  d  b
8   b  f  y
17  d  n  z
20  f  k  f
5   g  v  e
10  g  u  r
6   h  b  x
11  h  x  w
1   k  g  i
3   k  a  k
15  n  w  q
7   q  o  b
9   q  z  s
16  r  b  x
18  r  r  x
2   s  b  v
4   s  r  y
13  s  h  h
12  t  u  u
14  u  p  n
```

```
# also in decreasing order
x[order(x$v1, decreasing = TRUE), ]
```

```
   v1 v2 v3
14  u  p  n
```

```
12  t  u  u
2   s  b  v
4   s  r  y
13  s  h  h
16  r  b  x
18  r  r  x
7   q  o  b
9   q  z  s
15  n  w  q
1   k  g  i
3   k  a  k
6   h  b  x
11  h  x  w
5   g  v  e
10  g  u  r
20  f  k  f
17  d  n  z
8   b  f  y
19  a  d  b
```

`order` can take several vectors to do hierarchical sorting.

```
i <- order(x$v2, x$v1, x$v3)   (sort by v2, break ties with v1, then break ties by v3)
i
```

```
 [1]  3  6 16  2 19  8  1 13 20 17  7 14 18  4 10 12  5 15 11  9
```

```
x[i, ]
```

```
   v1 v2 v3
3   k  a  k
6   h  b  x
16  r  b  x
2   s  b  v
19  a  d  b
8   b  f  y
1   k  g  i
13  s  h  h
20  f  k  f
17  d  n  z
7   q  o  b
14  u  p  n
18  r  r  x
4   s  r  y
10  g  u  r
12  t  u  u
5   g  v  e
15  n  w  q
11  h  x  w
9   q  z  s
```

## Character and string manipulation

**nchar**

length - number of elements in a character vector
nchar - number of characters in each element

A character vector is a vector where every element is a character string of any length. The `length()` of a character vector is the number of elements in it:

```
x <- c("This is a sentence", "Hello World!", "This is the third element")
length(x)
```

```
[1] 3
```

To get the number of characters in each element, use `nchar()`:

```
nchar(x)
```

```
[1] 18 12 25
```

shown in the same order as they are in the character vector

**substr**

Strings can be extracted from elements using `substr()`. You specify the first and last characters to be extracted from each string:

```
# get the first three characters from every string
substr(x, 1, 3)
```

```
[1] "Thi" "Hel" "Thi"
```

```
# get the 3rd character from every string
substr(x, 3, 3)
```

```
[1] "i" "l" "i"
```

`substr` can also be used to replace values within strings by assigning:

```
substr(x, 1, 4) <- "That"
x
```

```
[1] "That is a sentence"        "Thato World!"
[3] "That is the third element"
```

### strsplit

Strings can be split based on some common delimiter using `strsplit()`:

```
# split based on spaces
x.split <- strsplit(x, " ")
x.split
```

```
[[1]]
[1] "That"     "is"       "a"        "sentence"

[[2]]
[1] "Thato"  "World!"

[[3]]
[1] "That"    "is"      "the"     "third"   "element"
```

```
str(x.split)
```

```
List of 3
 $ : chr [1:4] "That" "is" "a" "sentence"
 $ : chr [1:2] "Thato" "World!"
 $ : chr [1:5] "That" "is" "the" "third" ...
```

Note that the return value from `strsplit` is a list. Each element in the list corresponds to a vector resulting from splitting every element in the orginal vector

```
x.split[[1]]
```

```
[1] "That"     "is"       "a"        "sentence"
```

### paste

To create strings from combinations of strings (or numbers) we use `paste()`. This function takes a set of vectors, and pastes the elements together using recycling:

```
# vectors are equal length
paste(letters[1:6], 1:6)
```

```
[1] "a 1" "b 2" "c 3" "d 4" "e 5" "f 6"
```

```
# one vector is a multiple of the other
paste(letters[1:6], 1:2)
```

```
[1] "a 1" "b 2" "c 1" "d 2" "e 1" "f 2"
```

```
# one vector is not a multiple of the other
paste(letters[1:6], 1:4)
```

```
[1] "a 1" "b 2" "c 3" "d 4" "e 1" "f 2"
```

The argument `sep` determines what character is used as a separator between the characters:

```
paste(letters[1:6], 1:2, sep = "-")
```

```
[1] "a-1" "b-2" "c-1" "d-2" "e-1" "f-2"
```

If you do not want a separator character, either set `sep = ""` or use `paste0()`:

```
paste0(letters[1:6], 1:2)
```

```
[1] "a1" "b2" "c1" "d2" "e1" "f2"
```

If you want to paste all of the arguments to create a single element vector, set the `collapse` argument:

```
paste(letters[1:6], 1:2, sep = "-", collapse = "#")
```

```
[1] "a-1#b-2#c-1#d-2#e-1#f-2"
```

### tolower, toupper

Character case can be changed with `tolower` and `toupper`:

```
tolower(x)
```

```
[1] "that is a sentence"        "thato world!"
[3] "that is the third element"
```

```
toupper(x)
```

```
[1] "THAT IS A SENTENCE"        "THATO WORLD!"
[3] "THAT IS THE THIRD ELEMENT"
```

## Regular Expressions

For finer control on searching and replacing text within strings, you will have to turn to "regular expressions", which is a kind of syntax of its own and is common across several platforms. The help page for regular expressions in R is `?regex`. The functions that are most commonly used with regular expressions are given in `grep`. The most commonly used on this page are:

`grep` and `grepl`: Identify elements that have the sought after pattern `sub` and `gsub`: Replace a desired pattern with other text

```
x <- c("Here is some text", "This is more text", "I have the number 1", "22 is the number I have")
# which elements have the word "text"?
grep("text", x)
```

```
[1] 1 2
```

```
# which elements have numbers?
grep("[[:digit:]]", x)
```

```
[1] 3 4
```

```
# replace the word "This" with "That"
gsub("This", "That", x)
```

```
[1] "Here is some text"       "That is more text"
[3] "I have the number 1"     "22 is the number I have"
```

## apply Functions

Many times, we want to execute the same function on sequential elements of some object. This could be things like the elements of a vector or list, the rows of a matrix, or the columns of a data frame. For these, R provides a family of functions that usually end in `-apply` or are based on them.

### lapply

The most basic of these functions is `lapply`. The "l" refers to the fact that `lapply` will always return a list. There are two main arguments to `lapply`: the first is the object to be iterated over, and the second is a function that takes sequential elements of that object. As an example, let's use the `sample` function. Recall that if you execute `sample` with a single integer(`n`), it will return a permutation of the vector `1:n`:

```
sample(5)
```

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

```
sample(10)
```

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

Here is a list resulting from calls to sample with the elements of the vector `1:5`:

```
x <- lapply(1:5, sample)
str(x)
```

```
List of 5
 $ : int 1
 $ : int [1:2] 1 2
 $ : int [1:3] 3 2 1
 $ : int [1:4] 3 4 2 1
 $ : int [1:5] 1 4 3 5 2
```

```
x
```

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

[[2]]
[1] 1 2

[[3]]
[1] 3 2 1

[[4]]
[1] 3 4 2 1

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

Note that the result is a list, the elements of which are the result of calls to `sample(1)`, `sample(2)`, `sample(3)`, etc. The elements of the return value are in the same order as the original object being iterated over:

```
lapply(c(5, 3, 1, 8), sample)
```

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

[[2]]
[1] 1 3 2

[[3]]
[1] 1

[[4]]
[1] 4 2 7 6 3 5 8 1
```

The first argument can be a list too:

```
lapply(x, sum)
```

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

[[2]]
[1] 3

[[3]]
[1] 6

[[4]]
[1] 10
```

```
[[5]]
[1] 15
```

### sapply

If the return value from every iteration was the same length, you may want to simplify the result. This is what `sapply` is for. If every call to the function returns a scalar, then `sapply` will return a vector. If every call to the function returns a vector of equal length, then `sapply` will return a matrix. If every call to the function returns a value of different lengths, then `sapply` defaults to returning a list:

```
# every return value from sum is a scalar - sapply returns a vector
sapply(x, sum)
```

```
[1]  1  3  6 10 15
```

```
# every return value from sample is a 5 element vector - sapply returns a matrix
sapply(rep(5, 8), sample)
```

```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
[1,]    3    3    2    1    1    4    3    2
[2,]    4    2    1    3    5    2    5    1
[3,]    2    4    4    2    2    5    4    3
[4,]    1    5    5    4    3    3    1    4
[5,]    5    1    3    5    4    1    2    5
```

```
# this is the same as our lapply example - sapply returns a list
sapply(c(5, 3, 1, 8), sample)
```

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

[[2]]
[1] 1 2 3

[[3]]
[1] 1

[[4]]
[1] 8 3 2 7 6 4 5 1
```

Arguments to the function can be specified in the `lapply` or `sapply` call:

```
sapply(c(5, 3, 1, 8), sample, size = 5, replace = TRUE)
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    2    1    6
[2,]    4    1    1    7
[3,]    1    3    1    4
[4,]    4    1    1    8
[5,]    4    2    1    4
```

### apply

If you are dealing with a multi-dimensional object (matrix, array, or data frame) and you want to apply a function to a given dimension (i.e, each row or each column), use `apply`. You have to specify the dimension

that you will be iterating over as the second argument (1 = rows, 2 = columns, etc). `apply` will try to simplify the results like `sapply`:

```
x <- matrix(sample(1:100, 24, replace = TRUE), nrow = 4)
x
```

```
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]   27   81   83   65   90   28
[2,]    8   51   21    1   20    8
[3,]    1   43   67   85   72   78
[4,]  100   15   83   14   96   17
```

```
# median of each row
apply(x, 1, median)
```

```
[1] 73.0 14.0 69.5 50.0
```

```
# difference of each column
apply(x, 2, diff)
```

```
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]  -19  -30  -62  -64  -70  -20
[2,]   -7   -8   46   84   52   70
[3,]   99  -28   16  -71   24  -61
```

### tapply

If you want to execute a function on groups of values, `tapply` can often be a good choice. The arguments are a vector that will be summarized, another vector or set of vectors that represent identify elements to groups, and the function that will get the sequential subsets of the original vector. As a simple example, we calculate the means of subsets of a random vector of numbers

```
x <- sample(1:100, 100, replace = TRUE)
grp <- sample(letters[1:5], 100, replace = TRUE)
tapply(x, grp, mean)
```

```
       a        b        c        d        e
49.94444 55.76190 61.09091 48.27273 50.29412
```

As a more practical example, we can calculate the average temperature at each station in our ctd dataset:

```
ctd <- read.csv("ctd.csv")
tapply(ctd$temp, ctd$station, mean)
```

```
 Station.1 Station.10 Station.11 Station.12 Station.13 Station.14
  13.56772   14.57675   14.93466   14.40093   13.84903   14.42424
Station.15 Station.16 Station.17 Station.18 Station.19  Station.2
  14.10233   14.42280   14.27389   14.57620   16.32791   14.24618
Station.20 Station.21 Station.22 Station.23 Station.24 Station.25
  13.34375   13.77648   14.18930   14.60143   16.35819   15.82101
Station.26 Station.27 Station.28 Station.29  Station.3 Station.30
  15.84702   14.10141   13.21872   13.76510   14.23830   14.19600
Station.31 Station.32 Station.33 Station.34 Station.35 Station.36
  14.85625   16.73097   14.07543   14.62647   14.74482   15.53890
Station.37 Station.38 Station.39  Station.4 Station.40  Station.5
  15.12451   15.57174   15.14841   14.51093   16.37120   14.65009
 Station.6  Station.7  Station.8  Station.9
  14.23170   13.63123   13.94914   14.18727
```

We can use two grouping variables to return a matrix. However, when we do this, the second argument must be specified as a list.

```
# What is the average temperature at each station and depth?
mean.temp <- tapply(ctd$temp, list(station = ctd$station, depth = ctd$depth), mean)
head(mean.temp[, 1:5])
```

```
            depth
station             1        2        3        4        5
  Station.1  17.22627 17.18102 17.07373 16.92864 16.75797
  Station.10 16.67695 16.55271 16.24712 15.95271 15.66983
  Station.11 16.39310 16.17458 15.89458 15.59407 15.34814
  Station.12 16.86448 16.74119 16.52642 16.35448 16.15030
  Station.13 17.05638 16.86203 16.68034 16.49610 16.24271
  Station.14 16.98061 16.81493 16.59866 16.41940 16.16791
```

### aggregate

If we want to apply the same grouped summary to every column in a data frame, we can use `aggregate`:

```
# what is the median of each measurement at each station?
st.medians <- aggregate(ctd[, 3:8], list(station = ctd$station), median, na.rm = TRUE)
head(st.medians)
```

```
     station   temp salinity  dox   ph pct_light density
1  Station.1 13.070  33.4570 7.05 8.05    88.330 25.1380
2 Station.10 14.445  33.4695 7.90 8.15    81.530 24.8515
3 Station.11 14.940  33.4625 7.88 8.15    76.270 24.7710
4 Station.12 14.095  33.4530 7.66 8.12    85.035 24.8905
5 Station.13 13.500  33.4635 7.42 8.10    86.640 25.0325
6 Station.14 14.170  33.4640 7.67 8.13    84.700 24.8860
```

Be careful if the function returns more than one thing though.

```
st.range <- aggregate(ctd[, 3:8], list(station = ctd$station), range, na.rm = TRUE)
head(st.range)
```

```
     station temp.1 temp.2 salinity.1 salinity.2 dox.1 dox.2 ph.1 ph.2
1  Station.1   9.92  22.74     33.130     34.033  2.06 10.61 7.66 8.62
2 Station.10  10.36  22.65     33.162     33.864  2.14 13.03 7.66 8.55
3 Station.11  10.58  23.06     33.209     33.817  2.52 11.77 7.69 8.50
4 Station.12  10.24  23.00     32.561     34.311  2.28 11.38 7.67 8.63
5 Station.13  10.00  22.99     33.090     33.879  2.51 10.88 7.69 8.59
6 Station.14  10.20  22.74     33.069     33.891  2.25 11.35 7.65 8.61
  pct_light.1 pct_light.2 density.1 density.2
1       69.45       92.25    22.923    26.196
2       30.53       89.64    22.945    25.995
3        5.34       89.20    22.822    25.895
4       47.59       90.87    22.841    26.041
5       55.29       91.79    22.841    26.076
6       41.77       90.71    22.909    26.052
```

```
str(st.range)
```

```
'data.frame':   40 obs. of  7 variables:
 $ station  : Factor w/ 40 levels "Station.1","Station.10",..: 1 2 3 4 5 6 7 8 9 10 ...
 $ temp     : num [1:40, 1:2] 9.92 10.36 10.58 10.24 10 ...
```

```
 $ salinity : num [1:40, 1:2] 33.1 33.2 33.2 32.6 33.1 ...
 $ dox      : num [1:40, 1:2] 2.06 2.14 2.52 2.28 2.51 2.25 2.26 2.24 2.33 2.33 ...
 $ ph       : num [1:40, 1:2] 7.66 7.66 7.69 7.67 7.69 7.65 7.67 7.66 7.65 7.67 ...
 $ pct_light: num [1:40, 1:2] 69.45 30.53 5.34 47.59 55.29 ...
 $ density  : num [1:40, 1:2] 22.9 22.9 22.8 22.8 22.8 ...
```

Note that the column names seem to have `.1` and `.2` when you print the object, but they aren't in the structure. In this case, every measurement column is itself a two column matrix:

```
dim(st.range$temp)
```

```
[1] 40  2
```

```
head(st.range$temp)
```

```
      [,1]  [,2]
[1,]  9.92 22.74
[2,] 10.36 22.65
[3,] 10.58 23.06
[4,] 10.24 23.00
[5,] 10.00 22.99
[6,] 10.20 22.74
```

### by

To apply a function to an entire data frame, use `by()`, which is works much like `tapply`:

```
# How many records per station?
st.rows <- by(ctd, ctd$station, nrow)
head(st.rows)
```

```
ctd$station
 Station.1 Station.10 Station.11 Station.12 Station.13 Station.14
      3535       1120        762       1876       2229       1865
```

```
str(st.rows)
```

```
 'by' int [1:40(1d)] 3535 1120 762 1876 2229 1865 1826 1865 1474 1120 ...
 - attr(*, "dimnames")=List of 1
  ..$ ctd$station: chr [1:40] "Station.1" "Station.10" "Station.11" "Station.12" ...
 - attr(*, "call")= language by.data.frame(data = ctd, INDICES = ctd$station, FUN = nrow)
```

```
st.rows["Station.5"]
```

```
Station.5
      809
```

You can also summarize with multiple groups, which have to be included as a list:

```
# How many records per station?
st.depth.rows <- by(ctd, list(station = ctd$station, depth = ctd$depth), nrow)
str(st.depth.rows)
```

```
 'by' int [1:40, 1:60] 59 59 58 67 58 66 59 67 59 59 ...
 - attr(*, "dimnames")=List of 2
  ..$ station: chr [1:40] "Station.1" "Station.10" "Station.11" "Station.12" ...
  ..$ depth  : chr [1:60] "1" "2" "3" "4" ...
 - attr(*, "call")= language by.data.frame(data = ctd, INDICES = list(station = ctd$station, depth = ct
```

```
# The object can be indexed like a matrix
st.depth.rows["Station.1", "12"]
```

```
[1] 59
```

### mapply

To apply a function to sequential elements of multiple vectors, use `mapply`. The first argument is a function, and every argument afterwards is an argument to that function composed of vectors being iterated over. For example, the following creates a list of random numbers of alternating length with increasing range:

```
mapply(sample, x = 5:10, size = c(20, 4), replace = TRUE)
```

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

[[2]]
[1] 3 1 3 4

[[3]]
 [1] 4 7 6 3 2 3 4 1 1 7 3 1 3 5 5 5 5 1 5 4

[[4]]
[1] 8 6 3 5

[[5]]
 [1] 6 6 6 7 9 9 9 9 3 1 9 6 8 1 5 3 5 8 1 8

[[6]]
[1]  2 10  7  9
```

### split

A handy function for creating lists based on a grouping variable is `split`. It will split a vector, matrix, or data frame. For instance, here is a list where every element is a data frame containing only one station's data:

```
st.list <- split(ctd, ctd$station)
head(st.list[[1]])
```

```
    station sample_date  temp salinity  dox   ph pct_light density depth
1 Station.1  2012-11-08 16.81   33.420 8.07 8.20     90.32  24.346    16
2 Station.1  2012-04-19 10.52   33.805 3.16 7.73     88.14  25.930    18
3 Station.1  2010-01-06 15.11   33.415 7.22 8.13     88.97  24.725    32
4 Station.1  2014-02-06 14.00   33.430 7.31   NA     88.01  24.974    41
5 Station.1  2011-01-05 14.20   33.286 7.91 8.16     86.17  24.822     3
6 Station.1  2015-02-03 13.92   33.382 6.45 8.05     87.68  24.953    51
```

```
head(st.list[[2]])
```

```
        station sample_date  temp salinity  dox   ph pct_light density
3536 Station.10  2010-05-10 14.99   33.479 9.62 8.35     70.32  24.799
3537 Station.10  2011-02-02 13.10   33.337 7.24 8.06     65.39  25.085
3538 Station.10  2010-03-17 13.45   33.406 8.62 8.17     73.64  25.069
3539 Station.10  2016-08-02 19.91   33.465 8.98 8.28     82.14  23.616
3540 Station.10  2016-11-02 14.00   33.279 6.68 8.02     79.46  24.858
```

```
3541 Station.10  2010-03-17 13.53   33.404 8.62 8.20     72.08  25.050
     depth
3536     4
3537     6
3538     6
3539    12
3540    19
3541     4
```

Here's the same creating an elment for each cast (station x date):

```
st.dt.list <- split(ctd, list(station = ctd$station, date = ctd$sample_date))
st.dt.list[[1]]
```

```
[1] station     sample_date temp        salinity    dox         ph
[7] pct_light   density     depth
<0 rows> (or 0-length row.names)
```

Because it does all combinations of the grouping factors, a lot will be empty. Let's find them:

```
num.rows <- sapply(st.dt.list, nrow)
zero.rows <- which(num.rows == 0)
st.dt.list <- st.dt.list[-zero.rows]
st.dt.list[[1]]
```

```
        station sample_date  temp salinity  dox   ph pct_light density
5929 Station.12  2010-01-05 14.72   33.374 7.61 8.18     78.87  24.779
6294 Station.12  2010-01-05 14.72   33.374 7.61 8.18     79.14  24.778
6295 Station.12  2010-01-05 14.72   33.373 7.59 8.18     79.32  24.778
6750 Station.12  2010-01-05 14.72   33.373 7.59 8.18     79.11  24.778
6775 Station.12  2010-01-05 14.72   33.374 7.60 8.18     78.97  24.779
6778 Station.12  2010-01-05 14.64   33.375 7.50 8.18     77.96  24.796
6794 Station.12  2010-01-05 14.72   33.373 7.59 8.18     79.13  24.778
6856 Station.12  2010-01-05 14.71   33.374 7.56 8.18     79.24  24.781
6957 Station.12  2010-01-05 14.59   33.373 7.43 8.16     68.05  24.807
6973 Station.12  2010-01-05 14.59   33.373 7.44 8.16     69.51  24.806
6992 Station.12  2010-01-05 14.72   33.365 7.59 8.18     78.82  24.773
7061 Station.12  2010-01-05 14.71   33.363 7.62 8.19     78.36  24.772
7067 Station.12  2010-01-05 14.75   33.245 7.52 8.17     74.31  24.674
7087 Station.12  2010-01-05 14.72   33.368 7.58 8.18     79.09  24.775
7094 Station.12  2010-01-05 14.59   33.373 7.45 8.17     70.50  24.805
7103 Station.12  2010-01-05 14.60   33.374 7.45 8.17     72.98  24.804
7108 Station.12  2010-01-05 14.75   33.243 7.52 8.17     74.02  24.671
7131 Station.12  2010-01-05 14.73   33.328 7.61 8.18     76.89  24.742
7161 Station.12  2010-01-05 14.71   33.364 7.61 8.18     79.00  24.773
7172 Station.12  2010-01-05 14.71   33.360 7.64 8.18     78.11  24.770
7199 Station.12  2010-01-05 14.72   33.366 7.58 8.18     78.80  24.773
7218 Station.12  2010-01-05 14.72   33.347 7.63 8.18     77.19  24.759
7260 Station.12  2010-01-05 14.73   33.310 7.58 8.18     74.98  24.727
7266 Station.12  2010-01-05 14.72   33.372 7.59 8.18     78.90  24.777
7271 Station.12  2010-01-05 14.72   33.368 7.60 8.18     78.87  24.775
7275 Station.12  2010-01-05 14.74   33.268 7.55 8.17     73.94  24.693
7284 Station.12  2010-01-05 14.73   33.324 7.60 8.18     76.35  24.739
7285 Station.12  2010-01-05 14.72   33.371 7.58 8.18     79.13  24.777
     depth
5929    21
6294    20
```

```
6295    19
6750    18
6775    22
6778    24
6794    17
6856    23
6957    28
6973    27
6992    11
7061     9
7067     2
7087    14
7094    26
7103    25
7108     1
7131     6
7161    10
7172     8
7199    12
7218     7
7260     4
7266    16
7271    13
7275     3
7284     5
7285    15
```

## Function Definition

The basic format for declaring a function actually uses a function called (wait for it...) `function`. I like to think of every function as having four components:

- **Name** - The name a user will use to call the function.
- **Arguments** - The input values a function needs to operate on.
- **Body** - The code that processes the arguments.
- **Return Value** - The output from the result of the processing in **Body**.

However, in any given function, depending on its purpose, one or more of the above items may be missing. Here is a simple function that has all four components designed to determine if `x` is between `a` and `b`:

```
isBetween <- function(x, a, b) {
  gt.a <- x > a
  lt.b <- x < b
  btwn <- gt.a & lt.b
  return(btwn)
}
```

In this function, the **name** is `isBetween`, and if we call it with the **arguments** `x = 6`, `a = 2`, and `b = 10`, it will **return** the value `TRUE`:

```
isBetween(x = 6, a = 2, b = 10)
```

```
[1] TRUE
```

### Name

A function's name should be short(ish), but also meaningful and easy to understand. This is an art and you should take the time to play with names until they fit. Pretend like you are a naive user who has no idea what the function does. You should be able to get most of that information from the name. Also pay attention to how other functions that your function will be working with are named. For example, if you have a function that reads a particular data file and formats it, but is only one of several data files that will be read, it would be bad form to call that function `readData`. It might be better to call it something like, `readSalinityData`. Even better might be, `readAndFormatSalinityData`. However, it would be unnecessarily long and mean to users to call it something like, `readSalinityDataFromCSVFileAndRemoveMissingDataPoints`.

### Arguments

These are the input values that the function needs to operate on. It is good programming practice to make them both as short and as long as necessary to be descriptive. In general, names for arguments should also be short and descriptive. However, some argument names are frequently used, such as `x` for the first argument, and `y` for the second argument, especially in mathematically-based functions or for data for axes in plotting functions. **It is good practice to not refer to anything in the function body that is either not in the arguments, or is not created in the function body.**

### Body

This is the code that is the heart of the function. It operates on the arguments to convert them to a value to be returned or perform an action. Curly braces (`{` and `}`) are used to denote the code that composes the body and belongs to the function. If the function only has one line for a body then the curly braces can be omitted. For example:

```
isBetween.2 <- function(x, a, b) return(x > a & x < b)
isBetween.2(x = 6, a = 2, b = 10)
```

```
[1] TRUE
```

### Return Value

A function can only return one object. The function called `return` is often used to denote what this object is as in the above examples. However, if there is no call to `return`, then **the result of the last line in a function is its return value**. For example, our `addAndRaise` function could also be written as:

```
isBetween.3 <- function(x, a, b) {
  gt.a <- x > a
  lt.b <- x < b
  gt.a & lt.b
}
isBetween.3(x = 6, a = 2, b = 10)
```

```
[1] TRUE
```

Sometimes you want a function to do an action, but only return a value if it is assigned to something. In this case, use the `invisible` function. In this example, our `isBetween` function will not print the result when called by itself:

```
isBetween <- function(x, a, b) {
  gt.a <- x > a
  lt.b <- x < b
```

```
  invisible(gt.a & lt.b)
}
# nothing printed
isBetween(x = 6, a = 2, b = 10)

# assign to object
result <- isBetween(x = 6, a = 2, b = 10)
result
```

```
[1] TRUE
```

## Arguments

To better understand how arguments are handled, let's first create a function that abbreviates vectors of scientific names to shorter versions. For instance, we want a function that takes "Homo" and "sapiens" and creates "H sap":

```
abbrev <- function(genus, species) {
  # get the first character from genus names
  g <- substr(genus, 1, 1)
  # get the first three characters from the species names
  spp <- substr(species, 1, 3)
  # paste the two together and return the result
  paste(g, spp)
}
```

Let's also load some data to use with it:

```
spp.codes <- read.csv("tblCodeSpecies.csv", stringsAsFactors = FALSE)
head(spp.codes)
```

```
  SPCODE   ORDER   SUBORDER      FAMILY  FAMILY.NAMES      GENUS
1    001 CETACEA ODONTOCETI   ZIPHIIDAE BEAKED WHALES Mesoplodon
2    002 CETACEA ODONTOCETI DELPHINIDAE      DOLPHINS   Stenella
3    003 CETACEA ODONTOCETI DELPHINIDAE      DOLPHINS   Stenella
4    004 CETACEA ODONTOCETI DELPHINIDAE      DOLPHINS   Stenella
5    005 CETACEA ODONTOCETI DELPHINIDAE      DOLPHINS  Delphinus
6    006 CETACEA ODONTOCETI DELPHINIDAE      DOLPHINS   Stenella
              SPECIES                  COMMON.NAME
1          peruvianus           Pygmy beaked whale
2           attenuata  Pantropical spotted dolphin
3 longirostris subsp. unidentified spinner dolphin
4             clymene              Clymene dolphin
5                 sp.  Unidentified common dolphin
6 attenuata graffmani      Coastal spotted dolphin
```

```
gns <- spp.codes$GENUS
spp <- spp.codes$SPECIES
```

. . . and test it out:

```
gspp <- abbrev(genus = gns, species = spp)
head(gspp)
```

```
[1] "M per" "S att" "S lon" "S cly" "D sp." "S att"
```

### Arguments - Matching

Argument are matched according to two rules. First, arguments that are specifically named, like `genus = gns` are matched. Then any remaining unnamed arguments are matched based on the order in which they are found. This is simple to understand in our two argument function, which we can call as we have before, or like this:

```
# 'species' is not named
gspp <- abbrev(genus = gns, spp)
head(gspp)
```

```
[1] "M per" "S att" "S lon" "S cly" "D sp." "S att"
```

```
# 'genus' is not named
gspp <- abbrev(species = spp, gns)
head(gspp)
```

```
[1] "M per" "S att" "S lon" "S cly" "D sp." "S att"
```

If you know the name and order of the arguments, none of them have to be named as long as they are always supplied in the correct order. In many commonly used functions, this is normal for the first few arguments. So we would normally call this function like this:

```
gspp <- abbrev(gns, spp)
head(gspp)
```

```
[1] "M per" "S att" "S lon" "S cly" "D sp." "S att"
```

Let's add a third argument to specify the number of characters in the species name to demonstrate the name and order matching further:

```
abbrev <- function(genus, species, num.spp) {
  # get the first character from genus names
  g <- substr(genus, 1, 1)
  # get the 'num.spp' characters from the species names
  spp <- substr(species, 1, num.spp)
  # paste the two together and return the result
  paste(g, spp)
}
gspp <- abbrev(gns, spp, 3)
head(gspp)
```

```
[1] "M per" "S att" "S lon" "S cly" "D sp." "S att"
```

```
# we can also call it like this:
gspp <- abbrev(num.spp = 3, gns, spp)
head(gspp)
```

```
[1] "M per" "S att" "S lon" "S cly" "D sp." "S att"
```

```
# this is fine too:
gspp <- abbrev(gns, num.spp = 3, spp)
head(gspp)
```

```
[1] "M per" "S att" "S lon" "S cly" "D sp." "S att"
```

```
# but this will not produce the desired output:
gspp <- abbrev(num.spp = 3, spp, gns)
head(gspp)
```

```
[1] "p Mes" "a Ste" "l Ste" "c Ste" "s Del" "a Ste"
```

Argument names can also be abbreviated as long as the abbreviations are unique. Let's add a fourth argument specifying the number of characters in the genus name to return:

```
abbrev <- function(genus, species, num.g, num.spp) {
  # get the first 'num.g' characters from genus names
  g <- substr(genus, 1, num.g)
  # get the first 'num.spp' characters from the species names
  spp <- substr(species, 1, num.spp)
  # paste the two together and return the result
  paste(g, spp)
}
gspp <- abbrev(gns, spp, num.g = 1, num.spp = 3)
head(gspp)
```

```
[1] "M per" "S att" "S lon" "S cly" "D sp." "S att"
```

We can abbreviate `gns` as just `g` because no other arguments start with "g":

```
gspp <- abbrev(s = spp, g = gns, num.g = 1, num.spp = 3)
head(gspp)
```

```
[1] "M per" "S att" "S lon" "S cly" "D sp." "S att"
```

However if we want to abbreviate `num.spp`, the shortest we can make it is `num.s` because any shorter than that and you couldn't differentiate it from `num.g`:

```
gspp <- abbrev(s = spp, g = gns, num.g = 1, num.s = 3)
head(gspp)
```

```
[1] "M per" "S att" "S lon" "S cly" "D sp." "S att"
```

This will produce an error:

```
gspp <- abbrev(s = spp, g = gns, num = 1, 3)
```

```
Error in abbrev(s = spp, g = gns, num = 1, 3): argument 3 matches multiple formal arguments
```

### Arguments - Defaults

Sometimes it is useful to specify default values for arguments. This means that users do not have to enter the default values every time an argument is called, but they can be modified if need be. Default values are specified by setting them directly in the argument list:

```
abbrev <- function(genus, species, num.g = 1, num.spp = 3) {
  # get the first 'num.g' characters from genus names
  g <- substr(genus, 1, num.g)
  # get the first 'num.spp' characters from the species names
  spp <- substr(species, 1, num.spp)
  # paste the two together and return the result
  paste(g, spp)
}
gspp <- abbrev(gns, spp)
head(gspp)
```

```
[1] "M per" "S att" "S lon" "S cly" "D sp." "S att"
```

But they can be changed by specifying them by name or position in the function call:

```
# set num.g to 3
gspp <- abbrev(gns, spp, 3)
head(gspp)
```

```
[1] "Mes per" "Ste att" "Ste lon" "Ste cly" "Del sp." "Ste att"
```

```
# set num.spp to 1
gspp <- abbrev(gns, spp, num.spp = 1)
head(gspp)
```

```
[1] "M p" "S a" "S l" "S c" "D s" "S a"
```

#### Arguments - Ellipses

There are times when you want to be able to pass arguments on to functions within your function, but you don't want to have to specify all possible arguments for that function in your argument list. To solve this, you can use the ellipses or dot-dot-dot notation, `...`. Here we use them to pass on formatting arguments like `sep` and `collapse` to the `paste` function:

```
abbrev <- function(genus, species, num.g = 1, num.spp = 3, ...) {
  # get the first 'num.g' characters from genus names
  g <- substr(genus, 1, num.g)
  # get the first 'num.spp' characters from the species names
  spp <- substr(species, 1, num.spp)
  # paste the two together and return the result
  paste(g, spp, ...)
}
gspp <- abbrev(gns, spp, sep = ".")
head(gspp)
```

```
[1] "M.per" "S.att" "S.lon" "S.cly" "D.sp." "S.att"
```

## Disposable Functions

There are times when an argument to a function is another function. In these cases, the function being passed can be predefined or, if it is being defined only for this purpose, it can just be defined directly in the argument list. This latter method creates what is called an "anonymous" or "disposable" function. One place this is frequently done is with the 'apply family of functions where one function is applied to every element in an object and the results are collected and returned in a convenient form. For example, let's say that species code data.frame is actually a list of data.frames with one element per Family:

```
families <- split(spp.codes, spp.codes$FAMILY)
names(families)
```

```
 [1] ""               "BALAENIDAE"      "BALAENOPTERIDAE"
 [4] "CHELONIDAE"     "DELPHINIDAE"     "DERMOCHELYIDAE"
 [7] "ESCHRICTIDAE"   "INIIDAE"         "KOGIIDAE"
[10] "MONODONTIDAE"   "MUSTELIDAE"      "NEOBALAENIDAE"
[13] "ODOBENIDAE"     "OTARIIDAE"       "PHOCIDAE"
[16] "PHOCOENIDAE"    "PHYSETERIDAE"    "PLATANISTIDAE"
[19] "PONTOPORIIDAE"  "RHINCODONTIDAE"  "SIRENIA"
[22] "UNIDENTIFIED"   "URSIDAE"         "ZIPHIIDAE"
```

Now we want to know how many genera are in each Family. Let's first make a function that takes a data.frame and returns the number of unique genera in that data.frame:

```
num.genera <- function(df) length(unique(df$GENUS))
# Number of genera in entire data.frame
num.genera(spp.codes)
```

```
[1] 82
```

```
# Number of genera in Ziphiidae
num.genera(families$ZIPHIIDAE)
```

```
[1] 6
```

To get the number of genera in each family, we want to use `sapply` on the `families` list, where the second argument is the function we want to apply to every element in `families`:

```
sapply(families, num.genera)
```

```
                  BALAENIDAE BALAENOPTERIDAE      CHELONIDAE
           2               2               2               6
   DELPHINIDAE  DERMOCHELYIDAE    ESCHRICTIDAE         INIIDAE
          20               2               1               1
      KOGIIDAE    MONODONTIDAE      MUSTELIDAE   NEOBALAENIDAE
           1               2               1               1
    ODOBENIDAE       OTARIIDAE        PHOCIDAE     PHOCOENIDAE
           1               8               8               5
  PHYSETERIDAE   PLATANISTIDAE   PONTOPORIIDAE  RHINCODONTIDAE
           1               1               2               1
       SIRENIA    UNIDENTIFIED         URSIDAE       ZIPHIIDAE
           2               7               1               6
```

However, if we only need the `num.genera` function for the `sapply` then we don't have to predefine it. We can create the disposable function directly in the sapply argument list for the `FUN` argument, which is second (see `?sapply`):

```
sapply(families, function(df) length(unique(df$GENUS)))
```

```
                  BALAENIDAE BALAENOPTERIDAE      CHELONIDAE
           2               2               2               6
   DELPHINIDAE  DERMOCHELYIDAE    ESCHRICTIDAE         INIIDAE
          20               2               1               1
      KOGIIDAE    MONODONTIDAE      MUSTELIDAE   NEOBALAENIDAE
           1               2               1               1
    ODOBENIDAE       OTARIIDAE        PHOCIDAE     PHOCOENIDAE
           1               8               8               5
  PHYSETERIDAE   PLATANISTIDAE   PONTOPORIIDAE  RHINCODONTIDAE
           1               1               2               1
       SIRENIA    UNIDENTIFIED         URSIDAE       ZIPHIIDAE
           2               7               1               6
```

The disposable function may be more than one line in which case we just need to use curly braces to collect the statements. In this example the function returns the number of genera and the number of species. Make sure to include the closing curly brace for the function and the closing parentheses for the `sapply`:

```
sapply(families, function(df) {
  num.gen <- length(unique(df$GENUS))
  num.spp <- length(unique(df$SPECIES))
  c(num.gen = num.gen, num.spp = num.spp)
})
```

```
          BALAENIDAE BALAENOPTERIDAE CHELONIDAE DELPHINIDAE DERMOCHELYIDAE
num.gen 2          2               2          6          20               2
num.spp 1          3               9          7          45               2
        ESCHRICTIDAE INIIDAE KOGIIDAE MONODONTIDAE MUSTELIDAE
num.gen            1       1        1           2          1
num.spp            1       1        3           2          1
        NEOBALAENIDAE ODOBENIDAE OTARIIDAE PHOCIDAE PHOCOENIDAE
num.gen             1          1         8        8           5
num.spp             1          1        10       14           7
        PHYSETERIDAE PLATANISTIDAE PONTOPORIIDAE RHINCODONTIDAE SIRENIA
num.gen            1             1             2              1       2
num.spp            1             2             2              1       3
        UNIDENTIFIED URSIDAE ZIPHIIDAE
num.gen            7       1         6
num.spp            1       1        24
```

## Flow Control

Normally when code is executed, the "flow" proceeds in a linear fashion. The first line is executed, then the second, and so forth until the last line of code is reached. There can be situations where you want to direct this flow either in branching form: some piece of code is executed based on one condition, while another piece is executed based on another condition, or in a looping manner: the same code is executed repeatedly until some stopping criterion is reached (either number of iterations, or a condition is met). There are several functions that allow you to manage this flow control, the help for which can be found with `?Control`.

### Branching

There are three standard branching functions:

- `if(cond) cons.expr else alt.expr` : executes a set of code (`cons.expr`) if `cond` evaluates to `TRUE` or (optionally) alternative code (`alt.expr`) if it is `FALSE`.
- `ifelse(test, yes, no)` : returns elements from `yes` matching to elements in `test` that are `TRUE` and elements in `no` for elements in `test` that are `FALSE`.
- `switch(EXPR, ...)` : executes individual code for named or numbered values in `EXPR`.

The thing to remember is that `if` is used for a single branching event (when `else` is not used) or a bifurcating branch (when `else` is used) that is based on a single condition (one `T` or `F`). `ifelse` is used to return a vector that is the same length as the logical vector with one set of values of for `TRUE` elements and another set for `FALSE` elements. `switch` is used in places where you want different pieces of code run for different discrete values. This is usually preferred if there are more than two possible conditions.

### if

As an example of `if`, lets construct some checks of argument ranges in our species abbreviation code:

```
abbrev <- function(genus, species, num.g = 1, num.spp = 3, ...) {
  # 'num.g' must be 1 or greater
  if(num.g < 1) num.g <- 1
  # 'num.g' shouldn't be too big
  if(num.g > 3) num.g <- 3
  # get the first 'num.g' characters from genus names
  g <- substr(genus, 1, num.g)
  # get the first 'num.spp' characters from the species names
  spp <- substr(species, 1, num.spp)
```

```
  # paste the two together and return the result
  paste(g, spp, ...)
}
gspp <- abbrev(gns, spp, num.g = 0, sep = ".")
head(gspp)
```

```
[1] "M.per" "S.att" "S.lon" "S.cly" "D.sp." "S.att"
```

```
gspp <- abbrev(gns, spp, num.g = 10, sep = ".")
head(gspp)
```

```
[1] "Mes.per" "Ste.att" "Ste.lon" "Ste.cly" "Del.sp." "Ste.att"
```

Let's say that there are just two abbreviation formats, a short one like: "H.sap" and just the abbreviated genus: "H. sapiens". We control this with a simple argument, called `short`:

```
abbrev <- function(genus, species, short = T) {
  if(short) {
    g <- substr(genus, 1, 1)
    spp <- substr(species, 1, 3)
    g.spp <- paste(g, spp, sep = ".")
    return(g.spp)
  } else {
    g <- substr(genus, 1, 1)
    g.spp <- paste(g, species, sep = ". ")
    return(g.spp)
  }
}
# The short form
head(abbrev(gns, spp))
```

```
[1] "M.per" "S.att" "S.lon" "S.cly" "D.sp." "S.att"
```

```
# The longer form
head(abbrev(gns, spp, F))
```

```
[1] "M. peruvianus"          "S. attenuata"
[3] "S. longirostris subsp." "S. clymene"
[5] "D. sp."                 "S. attenuata graffmani"
```

We used `return(g.spp)` in order to make sure the function returns the result from the execution branch for each condition. We can simplify this code in several convenient ways. The first is based on the fact that the last line in the expression for each condition is the return value of the `if` function. So we can assign `g.spp` based on each branch then return it once at the end:

```
abbrev <- function(genus, species, short = T) {
  g.spp <- if(short) {
    g <- substr(genus, 1, 1)
    spp <- substr(species, 1, 3)
    paste(g, spp, sep = ".")
  } else {
    g <- substr(genus, 1, 1)
    paste(g, species, sep = ". ")
  }
  return(g.spp)
}
# The short form
```

```
head(abbrev(gns, spp))
```

```
[1] "M.per" "S.att" "S.lon" "S.cly" "D.sp." "S.att"
```

```
# The longer form
head(abbrev(gns, spp, F))
```

```
[1] "M. peruvianus"          "S. attenuata"
[3] "S. longirostris subsp." "S. clymene"
[5] "D. sp."                 "S. attenuata graffmani"
```

Also notice that we create `g <- substr(genus, 1, 1)` in each expression, so we can move that to the outside. Also, because **the result of the last line in a function is its return value** we can remove the `return(g.spp)` line:

```
abbrev <- function(genus, species, short = T) {
  g <- substr(genus, 1, 1)
  if(short) {
    spp <- substr(species, 1, 3)
    paste(g, spp, sep = ".")
  } else {
    paste(g, species, sep = ". ")
  }
}
# The short form
head(abbrev(gns, spp))
```

```
[1] "M.per" "S.att" "S.lon" "S.cly" "D.sp." "S.att"
```

```
# The longer form
head(abbrev(gns, spp, F))
```

```
[1] "M. peruvianus"          "S. attenuata"
[3] "S. longirostris subsp." "S. clymene"
[5] "D. sp."                 "S. attenuata graffmani"
```

### ifelse

The `ifelse` function returns a vector that is as long as its first argument and chooses from the corresponding `yes` and `no` vectors to fill the elements. As an example, here's a function that will create the abbreviation "H. sapiens", but if the species name is longer than 8 characters, it will abbreviate that too:

```
abbrev <- function(genus, species) {
  g <- substr(genus, 1, 1)
  spp <- ifelse(nchar(species) > 8, substr(species, 1, 8), species)
  paste(g, spp, sep = ". ")
}
head(abbrev(gns, spp))
```

```
[1] "M. peruvian" "S. attenuat" "S. longiros" "S. clymene"  "D. sp."
[6] "S. attenuat"
```

The expressions in `ifelse` can be multiple lines too, but must be wrapped by curly braces. This modification adds a "." to the end of the abbreviated species name:

```
abbrev <- function(genus, species) {
  g <- substr(genus, 1, 1)
  spp <- ifelse(
```

```
    nchar(species) > 8,
    {
      spp.sub <- substr(species, 1, 8)
      paste0(spp.sub, ".")
    },
    species
  )
  paste(g, spp, sep = ". ")
}
head(abbrev(gns, spp))
```

```
[1] "M. peruvian." "S. attenuat." "S. longiros." "S. clymene"
[5] "D. sp."       "S. attenuat."
```

#### switch

The final branching function is `switch` which allows us to choose one of a series of expressions to execute based on a numeric or character value. For example, our abbreviation code will have an argument, `type` that will allow for three formats: `short` = "H.sap", `medium` = "H. sapiens", and `long` = "Homo sapiens":

```
abbrev <- function(genus, species, type) {
  g <- substr(genus, 1, 1)
  # we only need an `if` statement to format the species
  spp <- if(type == "short") substr(species, 1, 3) else species
  # choose the pasting format based on `type`
  switch(type,
    short = paste0(g, ".", spp),
    medium = paste0(g, ". ", spp),
    long = paste(genus, species)
  )
}
# The short form
head(abbrev(gns, spp, "short"))
```

```
[1] "M.per" "S.att" "S.lon" "S.cly" "D.sp." "S.att"
```

```
# The medium form
head(abbrev(gns, spp, "medium"))
```

```
[1] "M. peruvianus"          "S. attenuata"
[3] "S. longirostris subsp." "S. clymene"
[5] "D. sp."                 "S. attenuata graffmani"
```

```
# The long form
head(abbrev(gns, spp, "long"))
```

```
[1] "Mesoplodon peruvianus"        "Stenella attenuata"
[3] "Stenella longirostris subsp." "Stenella clymene"
[5] "Delphinus sp."                "Stenella attenuata graffmani"
```

## Looping

There are three functions to control looping:

- `for(var in seq)` : Executes a set of code for a number of iterations equal to the length of `seq`. In each iteration `var` gets sequential values of `seq`.

- `while(cond) expr` : Executes a set of code as long as `cond` is `TRUE`.
- `repeat expr` : Repeats code. To stop looping, execute `break`.

**for**

With `for` we execute a set of code for each element in a vector and in each execution, a variable takes sequential values of that vector. In the below example, we calculate the first `n` values of the fibonacci series.

```
fib <- function(n) {
  x <- 0
  for(i in 2:n) {
    if(i == 2) {
      x[i] <- 1
    } else {
      x[i] <- x[i - 1] + x[i - 2]
    }
  }
  x
}
fib(10)
```

```
 [1]  0  1  1  2  3  5  8 13 21 34
```

A `for` loop can be stopped with the `break` function. For example, we will put in a function that stops the loop the first time a number greater than 50 is reached:

```
fib <- function(n) {
  x <- 0
  for(i in 2:n) {
    if(i == 2) {
      x[i] <- 1
    } else {
      x[i] <- x[i - 1] + x[i - 2]
    }
    if(x[i] > 50) break
  }
  x
}
fib(13)
```

```
 [1]  0  1  1  2  3  5  8 13 21 34 55
```

We can also force the `for` loop to iterate again before reaching the natural end of code in an iteration with the `next` command. In this example, we only print values greater than the number specified in `print.num`:

```
fib <- function(n, print.num = 10) {
  x <- 0
  for(i in 2:n) {
    if(i == 2) {
      x[i] <- 1
    } else {
      x[i] <- x[i - 1] + x[i - 2]
    }
    if(x[i] < print.num) next
    cat(i, " : ", x[i], "\n")
  }
```

```
  x
}
fib(15)
```

```
8  :  13
9  :  21
10  :  34
11  :  55
12  :  89
13  :  144
14  :  233
15  :  377
 [1]   0   1   1   2   3   5   8  13  21  34  55  89 144 233 377
```

### while

The `while` function is designed to repeat some code until a condition is met. If the condition is never met, the loop will continue indefinitely. The `break` and `next` commands will perform the same with function and `repeat`. In this example, we print the fibonacci series until the specified number is exceeded:

```
fib <- function(n) {
  first <- 0
  second <- 1
  cat(first, " ")
  while(second <= n) {
    cat(second, " ")
    new.val <- first + second
    first <- second
    second <- new.val
  }
}
fib(20)
```

```
0  1  1  2  3  5  8  13
```

### repeat

The `repeat` function will continuously execute code until it is stopped by a `break`. Here, we do the same loop as above, but replace `while` with a `repeat` and `break`:

```
fib <- function(n) {
  first <- 0
  second <- 1
  cat(first, " ")
  repeat {
    cat(second, " ")
    new.val <- first + second
    if(new.val > n) break
    first <- second
    second <- new.val
  }
}
fib(20)
```

```
0  1  1  2  3  5  8  13
```

## Argument error checking

Despite our best efforts, functions are susceptible to users entering improper arguments or errors showing up that keeps a function from completing. We have some tools at our disposal to mitigate this. First, we can check that arguments what we expect and require using `if` statements. If they're aren't we can do something like return `NA` or `NULL`:

```
addTwo <- function(a, b) {
  # confirm that a and b are numbers
  if(!(is.numeric(a) | is.numeric(b))) return(NULL)
  a + b
}
addTwo(1, "x")
```

```
Error in a + b: non-numeric argument to binary operator
```

```
addTwo(1, 2)
```

```
[1] 3
```

We can also issue warnings when something unexpected happens:

```
addTwo <- function(a, b) {
  # confirm that a and b are numbers
  if(!(is.numeric(a) | is.numeric(b))) {
    warning("'a' and 'b' must be numbers. NULL returned.")
    return(NULL)
  }
  a + b
}
addTwo(1, "x")
```

```
Error in a + b: non-numeric argument to binary operator
```

If execution cannot or should not continue, then an error can be thrown with the `stop` function:

```
divideTwo <- function(a, b) {
  # confirm that a and b are numbers
  if(!(is.numeric(a) | is.numeric(b))) {
    stop("'a' and 'b' must be numbers. NULL returned.")
  }
  if(b == 0) {
    stop("'b' cannot be 0")
  }
  a / b
}
divideTwo(1, "x")
```

```
Error in a/b: non-numeric argument to binary operator
```

```
divideTwo(5, 0)
```

```
Error in divideTwo(5, 0): 'b' cannot be 0
```

## Scope

It is important to note that objects declared in a function only exist within that function. On the flipside, objects declared outside of a function are accessible by that function. However, it is very bad form to refer to an object in a function that has not been passed as an argument or declared in the function itself. For example:

```
a <- 2

my.func <- function(x, y) (x + y) * a

# this works
my.func(2, 3)
```

```
[1] 10
```

```
# remove a from the workspace
rm(a)

# a can't be found, so this produces an error
my.func(2, 3)
```

```
Error in my.func(2, 3): object 'a' not found
```

---

## Homework

Answer all questions in a script (.R) file. Use comments (`#` or `#'`) to explain steps in code.

1. In the "tblCodeSpecies.csv" data set, how many named species of the family "Balaenopteridae" are there?
2. In the "tblCodeSpecies.csv" data set, what is the mean number of species per family of Cetaceans (order Cetacea) and Pinnipeds (suborder Pinnipedia)?
3. In the "ctd.csv" data set, what is the mean difference in temperature between 10 meters and the surface?
4. In the "ctd.csv" data set, which stations have the lowest and highest mean surface temperature?
5. Write a function that returns the following summary statistics for a numerical vector: mean, standard deviation, number of values, number missing, minimum, maximum, and median.
6. Add to the function in 5 some code to make sure that the input is valid numerical vector.
7. Write a function that uses the function in 6 to summarize the measurements from a data.frame like "ctd.csv"
8. Write a function that uses the function in 6 to summarize the measurements from a CTD data.frame by station and depth.
9. Write a function that uses the function in 8 to identify outliers ($>$ 3 standard deviations from mean). The function should take as input a data.frame of CTD casts like "ctd.csv", a depth value of interest, and a measurement of interest. The output should be a three column data.frame that contains the station, sample date, and value of that measurement for every outlier identified.
10. Write a function that uses the function in 9 to identify outliers for every depth and measurement.