Stat 33B - Final Review

September 6, 2020

Vectors

R has no concept of scalars or arrays.

R's atomic data type is the **vector**, an ordered container for 0 or more elements.

Vector elements must all have the same data type.

The `c()` function combines vectors:

```
x = c(5, 7, 1)
x
```

```
## [1] 5 7 1
```

```
c(x, 1)
```

```
## [1] 5 7 1 1
```

```
c("hi", "hello")
```

```
## [1] "hi"    "hello"
```

```
"hi"
```

```
## [1] "hi"
```

```
c("hi", 1)
```

```
## [1] "hi" "1"
```

Vectorization

A **vectorized** function is one that is applied element-by-element when passed a vector argument.

Many R functions are vectorized:

```
c(sin(0), sin(1), sin(2))
```

```
## [1] 0.0000000 0.8414710 0.9092974
```

```
x = c(0, 1, 2)
sin(x)
```

```
## [1] 0.0000000 0.8414710 0.9092974
```

```
# NOT VECTORIZED:
mean(x)
```

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

Vectorization is the fastest kind of iteration in R.

Indexing

In R, indexes start at 1.

Use the square bracket `[` to access elements of a vector:

```
x = c(1, 3, 7)
x[2]
```

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

```
x[6]
```

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

You can use a vector as an index:

```
x[c(1, 1, 2)]
```

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

More about Data Frames

R uses data frames to represent tabular data.

A data frame is a list of column vectors. So:

- Elements of a column must all have the same type (like a vector).
- Elements of a row can have different types (like a list).
- Every row must be the same length.

In addition, every column must be the same length.

This idea is reflected in the type of a data frame:

```
data("iris")
typeof(iris)
```

```
## [1] "list"
```

Deconstructing Data Frames

The `unclass()` function resets the class of an object to match the object's type.

You can use `unclass()` to inspect the internals of an object.

For example, you can see that a data frame is a list:

```
unclass(iris)
```

```
## $Sepal.Length
##   [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7 5.4 5.1
##  [19] 5.7 5.1 5.4 5.1 4.6 5.1 4.8 5.0 5.0 5.2 5.2 4.7 4.8 5.4 5.2 5.5 4.9 5.0
##  [37] 5.5 4.9 4.4 5.1 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0 7.0 6.4 6.9 5.5
##  [55] 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6 6.7 5.6 5.8 6.2 5.6 5.9 6.1
##  [73] 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7 5.5 5.5 5.8 6.0 5.4 6.0 6.7 6.3 5.6 5.5
##  [91] 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7 6.3 5.8 7.1 6.3 6.5 7.6 4.9 7.3
## [109] 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5 7.7 7.7 6.0 6.9 5.6 7.7 6.3 6.7 7.2
## [127] 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3 6.1 7.7 6.3 6.4 6.0 6.9 6.7 6.9 5.8 6.8
## [145] 6.7 6.7 6.3 6.5 6.2 5.9
```

```
##
## $Sepal.Width
##   [1] 3.5 3.0 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 3.7 3.4 3.0 3.0 4.0 4.4 3.9 3.5
##  [19] 3.8 3.8 3.4 3.7 3.6 3.3 3.4 3.0 3.4 3.5 3.4 3.2 3.1 3.4 4.1 4.2 3.1 3.2
##  [37] 3.5 3.6 3.0 3.4 3.5 2.3 3.2 3.5 3.8 3.0 3.8 3.2 3.7 3.3 3.2 3.2 3.1 2.3
##  [55] 2.8 2.8 3.3 2.4 2.9 2.7 2.0 3.0 2.2 2.9 2.9 3.1 3.0 2.7 2.2 2.5 3.2 2.8
##  [73] 2.5 2.8 2.9 3.0 2.8 3.0 2.9 2.6 2.4 2.4 2.7 2.7 3.0 3.4 3.1 2.3 3.0 2.5
##  [91] 2.6 3.0 2.6 2.3 2.7 3.0 2.9 2.9 2.5 2.8 3.3 2.7 3.0 2.9 3.0 3.0 2.5 2.9
## [109] 2.5 3.6 3.2 2.7 3.0 2.5 2.8 3.2 3.0 3.8 2.6 2.2 3.2 2.8 2.8 2.7 3.3 3.2
## [127] 2.8 3.0 2.8 3.0 2.8 3.8 2.8 2.8 2.6 3.0 3.4 3.1 3.0 3.1 3.1 3.1 2.7 3.2
## [145] 3.3 3.0 2.5 3.0 3.4 3.0
##
## $Petal.Length
##   [1] 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 1.5 1.6 1.4 1.1 1.2 1.5 1.3 1.4
##  [19] 1.7 1.5 1.7 1.5 1.0 1.7 1.9 1.6 1.6 1.5 1.4 1.6 1.6 1.5 1.5 1.4 1.5 1.2
##  [37] 1.3 1.4 1.3 1.5 1.3 1.3 1.3 1.6 1.9 1.4 1.6 1.4 1.5 1.4 4.7 4.5 4.9 4.0
##  [55] 4.6 4.5 4.7 3.3 4.6 3.9 3.5 4.2 4.0 4.7 3.6 4.4 4.5 4.1 4.5 3.9 4.8 4.0
##  [73] 4.9 4.7 4.3 4.4 4.8 5.0 4.5 3.5 3.8 3.7 3.9 5.1 4.5 4.5 4.7 4.4 4.1 4.0
##  [91] 4.4 4.6 4.0 3.3 4.2 4.2 4.2 4.3 3.0 4.1 6.0 5.1 5.9 5.6 5.8 6.6 4.5 6.3
## [109] 5.8 6.1 5.1 5.3 5.5 5.0 5.1 5.3 5.5 6.7 6.9 5.0 5.7 4.9 6.7 4.9 5.7 6.0
## [127] 4.8 4.9 5.6 5.8 6.1 6.4 5.6 5.1 5.6 6.1 5.6 5.5 4.8 5.4 5.6 5.1 5.1 5.9
## [145] 5.7 5.2 5.0 5.2 5.4 5.1
##
## $Petal.Width
##   [1] 0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 0.2 0.2 0.1 0.1 0.2 0.4 0.4 0.3
##  [19] 0.3 0.3 0.2 0.4 0.2 0.5 0.2 0.2 0.4 0.2 0.2 0.2 0.2 0.2 0.4 0.1 0.2 0.2 0.2
##  [37] 0.2 0.1 0.2 0.2 0.3 0.3 0.2 0.6 0.4 0.3 0.2 0.2 0.2 0.2 1.4 1.5 1.5 1.3
##  [55] 1.5 1.3 1.6 1.0 1.3 1.4 1.0 1.5 1.0 1.4 1.3 1.4 1.5 1.0 1.5 1.1 1.8 1.3
##  [73] 1.5 1.2 1.3 1.4 1.4 1.7 1.5 1.0 1.1 1.0 1.2 1.6 1.5 1.6 1.5 1.3 1.3 1.3
##  [91] 1.2 1.4 1.2 1.0 1.3 1.2 1.3 1.3 1.1 1.3 2.5 1.9 2.1 1.8 2.2 2.1 1.7 1.8
## [109] 1.8 2.5 2.0 1.9 2.1 2.0 2.4 2.3 1.8 2.2 2.3 1.5 2.3 2.0 2.0 1.8 2.1 1.8
## [127] 1.8 1.8 2.1 1.6 1.9 2.0 2.2 1.5 1.4 2.3 2.4 1.8 1.8 2.1 2.4 2.3 1.9 2.3
## [145] 2.5 2.3 1.9 2.0 2.3 1.8
##
## $Species
##   [1] setosa     setosa     setosa     setosa     setosa     setosa
##   [7] setosa     setosa     setosa     setosa     setosa     setosa
##  [13] setosa     setosa     setosa     setosa     setosa     setosa
##  [19] setosa     setosa     setosa     setosa     setosa     setosa
##  [25] setosa     setosa     setosa     setosa     setosa     setosa
##  [31] setosa     setosa     setosa     setosa     setosa     setosa
##  [37] setosa     setosa     setosa     setosa     setosa     setosa
##  [43] setosa     setosa     setosa     setosa     setosa     setosa
##  [49] setosa     setosa     versicolor versicolor versicolor versicolor
##  [55] versicolor versicolor versicolor versicolor versicolor versicolor
##  [61] versicolor versicolor versicolor versicolor versicolor versicolor
##  [67] versicolor versicolor versicolor versicolor versicolor versicolor
##  [73] versicolor versicolor versicolor versicolor versicolor versicolor
##  [79] versicolor versicolor versicolor versicolor versicolor versicolor
##  [85] versicolor versicolor versicolor versicolor versicolor versicolor
##  [91] versicolor versicolor versicolor versicolor versicolor versicolor
##  [97] versicolor versicolor versicolor versicolor virginica  virginica
## [103] virginica  virginica  virginica  virginica  virginica  virginica
## [109] virginica  virginica  virginica  virginica  virginica  virginica
```

```
## [115] virginica  virginica  virginica  virginica  virginica  virginica
## [121] virginica  virginica  virginica  virginica  virginica  virginica
## [127] virginica  virginica  virginica  virginica  virginica  virginica
## [133] virginica  virginica  virginica  virginica  virginica  virginica
## [139] virginica  virginica  virginica  virginica  virginica  virginica
## [145] virginica  virginica  virginica  virginica  virginica  virginica
## Levels: setosa versicolor virginica
##
## attr(,"row.names")
##   [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
##  [19]  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
##  [37]  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
##  [55]  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
##  [73]  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
##  [91]  91  92  93  94  95  96  97  98  99 100 101 102 103 104 105 106 107 108
## [109] 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126
## [127] 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144
## [145] 145 146 147 148 149 150
```

Special Values

R has four special values.

Missing Values

`NA` represents a **missing value** in a data set:

```
NA
```

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

```
class(NA)
```

```
## [1] "logical"
```

```
typeof(NA)
```

```
## [1] "logical"
```

```
x = c(1, NA, 2)
class(x)
```

```
## [1] "numeric"
```

```
x = c("hi", NA)
class(x)
```

```
## [1] "character"
```

```
x
```

```
## [1] "hi" NA
```

```
c("NA", NA)
```

```
## [1] "NA" NA
```

The missing value `NA` is contagious!

```
NA + 3
```

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

```
3 * NA
```

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

```
7 / NA
```

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

Using a unknown argument in a computation usually produces an unknown result.

Null

`NULL` represents a value that's not defined *in R*.

`NULL` usually indicates absence of a result:

```
c()
```

```
## NULL
```

```
dim(c(1, 2))
```

```
## NULL
```

```
class(NULL)
```

```
## [1] "NULL"
```

```
typeof(NULL)
```

```
## [1] "NULL"
```

```
c(1, NULL)
```

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

For instance, if we try to get the matrix dimensions of a vector.

Not a Number

`NaN`, or "not a number", represents a value that's not defined mathematically.

```
NaN
```

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

```
class(NaN)
```

```
## [1] "numeric"
```

```
typeof(NaN)
```

```
## [1] "double"
```

```
c(1, NaN)
```

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

```
0 / 0
```

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

Infinite Values

`Inf` represents infinity. Produced by some computations:

```
13 / 0
```

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

```
Inf
```

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

```
c(1, Inf)
```

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

```
class(Inf)
```

```
## [1] "numeric"
```

```
typeof(Inf)
```

```
## [1] "double"
```

Three Ways to Subset

Use the subset (or "square bracket") operator `[` to get elements from a vector by position:

```
x = rep(c(4.1, 2.2, 7.9), 2)
x
```

```
## [1] 4.1 2.2 7.9 4.1 2.2 7.9
```

```
x[3]
```

```
## [1] 7.9
```

```
x[c(5, 4, 5)]
```

```
## [1] 2.2 4.1 2.2
```

You can also use the subset operator to set elements:

```
x[1] = 10


x
```

```
## [1] 10.0  2.2  7.9  4.1  2.2  7.9
```

```
x[c(4, 5)] = c(-20, -10)
x
```

```
## [1]  10.0   2.2   7.9 -20.0 -10.0   7.9
```

Negative positions mean "everything except":

```
x[-1]
```

```
## [1]   2.2   7.9 -20.0 -10.0   7.9
```

```
x[-c(5, 6)]
```

```
## [1]  10.0   2.2   7.9 -20.0
```

```
# x[c(5, -1)]
```

The subset operator `[` can actually get/set elements in three ways:

- By position
- By name
- By condition

Subsets by Name

You can make vectors with named elements (just like lists):

```
x = c(a = 6, b = 4, c = 3)
x
```

```
## a b c
## 6 4 3
```

```
names(x)
```

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

```
names(x) = c("hi", "hello", "goodbye")
x
```

```
##      hi   hello goodbye
##       6       4       3
```

You can use the subset operator `[` to get elements by name:

```
x["hello"]
```

```
## hello
##     4
```

```
x[c('hi', 'hi', 'goodbye')]
```

```
##      hi      hi goodbye
##       6       6       3
```

Likewise to set elements by name:

```
x["hello"] = 21

x
```

```
##      hi   hello goodbye
##       6      21       3
```

Congruent Vectors

Two vectors are **congruent** if they have the same length and they correspond elementwise.

For example, suppose you do a survey that records each person's:

- Favorite animal
- Age

These are two different vectors of information, but each person will have a response for both.

So you'll have two vectors that are the same length, with corresponding elements:

```
animal = c("dog", "cat", "iguana")
age = c(31, 24, 72)
```

These vectors are congruent.

Columns in a data frame are always congruent!

Subsets by Condition

The third way to subset a vector with `[` is to use a congruent logical vector.

For example:

```
x = c(2.2, 3.1, 6.7)
x
```

```
## [1] 2.2 3.1 6.7
```

```
logic = c(TRUE, FALSE, TRUE)

x[logic]
```

```
## [1] 2.2 6.7
```

`TRUE` means keep the element, `FALSE` means drop the element.

Missing values `NA` in the logical vector are retained in the result:

```
x[c(TRUE, NA, NA)]
```

```
## [1] 2.2  NA  NA
```

A **condition** is any expression that returns a logical vector.

For example, comparisons are conditions:

```
x > 3
```

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

You can use conditions to take subsets:

```
x[x > 3]
```

```
## [1] 3.1 6.7
```

Final note: you can technically subset with any logical vector.

If the length is different, R uses the recycling rule:

```
x[c(TRUE, FALSE)] #TRUE
```

```
## [1] 2.2 6.7
```

Tidy Data

Most Tidyverse packages, including `ggplot`, are designed for working with tidy data sets.

A data set is **tidy** if (and only if):

1. Each observation has its own row.
2. Each feature has its own column.
3. Each value has its own cell.

Tidy data sets are convenient for data analysis in general.

The `tidyr` package has tools to clean up untidy data sets, and also examples of untidy data.

Apply Function Basics

Doing the same operation repeatedly is a common pattern in programming.

Vectorization is one way, but not all functions are vectorized.

In R, the "apply functions" are another way to do something repeatedly.

The apply functions call a function on each element of a vector or list.

The `lapply()` Function

The first and most important apply function is `lapply()`. The syntax is:

```
lapply(X, FUN, ...)
```

The function `FUN` is called once for each element of `X`, with the element as the first argument. The `...` is for additional arguments to `FUN`, which are held constant across all calls.

Unrealistic example:

```
x = c(1, 7, 9)
lapply(x, sin)
```

```
## [[1]]
## [1] 0.841471
##
## [[2]]
## [1] 0.6569866
##
## [[3]]
## [1] 0.4121185
```

```
sin(x)
```

```
## [1] 0.8414710 0.6569866 0.4121185
```

In practice, it's clearer and more efficient to use vectorization here.

Let's use the iris data for some realistic examples:

```
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
```

```
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

```
lapply(iris, class)
```

```
## $Sepal.Length
## [1] "numeric"
##
## $Sepal.Width
## [1] "numeric"
##
## $Petal.Length
## [1] "numeric"
##
## $Petal.Width
## [1] "numeric"
##
## $Species
## [1] "factor"
```

```
class(iris)
```

```
## [1] "data.frame"
```

```
str(iris)
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

```
cols = c("Sepal.Width", "Petal.Length", "Petal.Width")
lapply(iris[cols], median, na.rm = TRUE)
```

```
## $Sepal.Width
## [1] 3
##
## $Petal.Length
## [1] 4.35
##
## $Petal.Width
## [1] 1.3
```

`lapply()` always returns the result as a list.

"l" for **list** result.

The `sapply()` Function

`sapply()` simplifies the result to a vector, when possible.

"s" for **simplified** result.

Examples:

```
sapply(iris[cols], median, na.rm = TRUE)
```

```
##  Sepal.Width Petal.Length  Petal.Width
##         3.00         4.35         1.30
```

The `sapply()` function is useful if you are working interactively.

The Split-Apply Strategy

The `split()` function splits a vector or data frame into groups based on some other vector (usually congruent).

```
x = c(1, 7, 9, 2, 5)
group = c("blue", "red", "blue", "green", "red")

split(x, group)
```

```
## $blue
## [1] 1 9
##
## $green
## [1] 2
##
## $red
## [1] 7 5
```

Split iris by the petal length column:

```
by_group = split(iris, iris$Petal.Length)
```

The `split()` function is especially useful when combined with `lapply()` or `sapply()`.

```
width_by_length = split(iris$Sepal.Width, iris$Petal.Length)
sapply(width_by_length, mean, na.rm = TRUE)
```

```
##        1      1.1      1.2      1.3      1.4      1.5      1.6      1.7
## 3.600000 3.000000 3.600000 3.228571 3.353846 3.569231 3.342857 3.600000
##      1.9        3      3.3      3.5      3.6      3.7      3.8      3.9
## 3.600000 2.500000 2.350000 2.300000 2.900000 2.400000 2.400000 2.633333
##        4      4.1      4.2      4.3      4.4      4.5      4.6      4.7
## 2.480000 2.833333 2.900000 2.900000 2.750000 2.875000 2.900000 3.060000
##      4.8      4.9        5      5.1      5.2      5.3      5.4      5.5
## 2.950000 2.820000 2.550000 2.875000 3.000000 2.950000 3.250000 3.033333
##      5.6      5.7      5.8      5.9        6      6.1      6.3      6.4
## 2.933333 3.266667 2.833333 3.100000 3.250000 3.133333 2.900000 3.800000
##      6.6      6.7      6.9
## 3.000000 3.300000 2.600000
```

This is an R idiom!

The `tapply()` Function

The `tapply()` function is equivalent to the `split()` and `sapply()` idiom.

"t" for **table**, because `tapply()` is a generalization of the frequency-counting function `table()`.

Examples:

```
tapply(iris$Sepal.Width, iris$Petal.Length, mean, na.rm = TRUE)
```

```
##        1      1.1      1.2      1.3      1.4      1.5      1.6      1.7
## 3.600000 3.000000 3.600000 3.228571 3.353846 3.569231 3.342857 3.600000
##      1.9        3      3.3      3.5      3.6      3.7      3.8      3.9
## 3.600000 2.500000 2.350000 2.300000 2.900000 2.400000 2.400000 2.633333
##        4      4.1      4.2      4.3      4.4      4.5      4.6      4.7
## 2.480000 2.833333 2.900000 2.900000 2.750000 2.875000 2.900000 3.060000
##      4.8      4.9        5      5.1      5.2      5.3      5.4      5.5
## 2.950000 2.820000 2.550000 2.875000 3.000000 2.950000 3.250000 3.033333
##      5.6      5.7      5.8      5.9        6      6.1      6.3      6.4
## 2.933333 3.266667 2.833333 3.100000 3.250000 3.133333 2.900000 3.800000
##      6.6      6.7      6.9
## 3.000000 3.300000 2.600000
```

```
# A generalization of table:
tapply(iris$Sepal.Width, iris$Sepal.Width, length)
```

```
##   2 2.2 2.3 2.4 2.5 2.6 2.7 2.8 2.9   3 3.1 3.2 3.3 3.4 3.5 3.6 3.7 3.8 3.9   4
##   1   3   4   3   8   5   9  14  10  26  11  13   6  12   6   4   3   6   2   1
## 4.1 4.2 4.4
##   1   1   1
```

```
table(iris$Sepal.Width)
```

```
##
##   2 2.2 2.3 2.4 2.5 2.6 2.7 2.8 2.9   3 3.1 3.2 3.3 3.4 3.5 3.6 3.7 3.8 3.9   4
##   1   3   4   3   8   5   9  14  10  26  11  13   6  12   6   4   3   6   2   1
## 4.1 4.2 4.4
##   1   1   1
```

This strategy is important for analyzing tabular data regardless of what programming language or packages you're using.

Developing Iterative Code

When thinking about writing a loop, try (in order):

1. vectorization
2. apply functions
   - Try an apply function if iterations are independent.
3. for/while-loops
   - Try a for-loop if some iterations depend on others.
   - Try a while-loop if the number of iterations is unknown.
4. recursion
   - Convenient for naturally recursive problems (like Fibonacci), but often there are faster solutions.

Handling Warnings and Errors

Use the `try` function to try running an expression that might produce an error:

```
x = try(5 + 6)

y = try(5 + "hi")
```

```
## Error in 5 + "hi" : non-numeric argument to binary operator
```

```
if (inherits(y, "try-error")) {
  # Handle error
} else {
  # Proceed normally
}
```

```
## NULL
```

If the expression produces an error, `try` returns an object with class `try-error`. Otherwise, it returns the result.

An error in `try` does NOT stop evaluation:

```
f = function(x, y) {
  try(5 + "hi")
  x + y
}
```

The error can be silenced by setting `silent = TRUE`:

```
f = function(x, y) {
  try(5 + "hi", silent = TRUE)
  x + y
}
```

Creating Closures

As an example, let's make a function that returns the number of times it's been called.

Here's the code:

```
counter = 0
count = function() {
  counter <<- counter + 1
  counter
}
```

Modifying the enclosing environment is a *side effect*.

Functions with side effects make code harder to understand and predict.

Use side effects sparingly. Most functions should not have side effects.

If you do need side effects, try to isolate them.

The example function has side effects on the global environment.

This extremely bad design!

The function might overwrite the user's variables:

```
counter = 0
count()
```

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

Or the user might overwrite the function's variables:

It's better to create a isolated enclosing environment for the function.

There are two different idioms for doing this:

1. Define the function inside of another function (a *factory function*)
2. Define the function inside of a call to `local`

Here's skeleton code for the first approach:

```
make_f = function() {
  # Put variables that `f` remembers between calls here:

  # This is your closure `f`:
  function() {
    # ...
  }
}

f = make_f()
# Now you can use f() as you would any other function.
```

We can use this approach for the example function:

```
make_count = function() {
  counter = 0

  function() {
    counter <<- counter + 1
    counter
  }
}

count = make_count()
```

Relational Data

Data split across multiple tables are called *relational data*.

A column shared by several tables is called a *key*.

For example, a grocery store's inventory system might have:

- A table that lists stores
- A table that lists items (fruits, vegetables, etc)
- A table that lists quantity of each item at each store

A *join* combines two separate tables by based on a common key.

For each row in the first table, the key is compared against the rows in the second table.

If the key matches, the rows are combined into a new row.

Note that a join can duplicate rows!

Four kinds of joins are:

- Inner: keep matches
- Full: keep everything
- Left: keep all left, keep matching right
- Right: keep matching left, keep all right

Joins are not just used in R; joins are especially important to understand for working with databases and SQL.