CSC 121: Computer Science for Statistics

Radford M. Neal, University of Toronto, 2017

http://www.cs.utoronto.ca/∼radford/csc121/

Week 6

Random Numbers and Their Uses

Random variation is a big part of what statistics is about. So it's natural that R has facilities to create its own random variation — to generate *random numbers*.

Random numbers have many uses (and not just in statistics):

- Simulate random processes, such as how a disease epidemic might spread between people.
- See how the results of some statistical method vary when the data it is applied to vary randomly.
- Compute things using "Monte Carlo" methods.
- Make interactions with a user have a random aspect — we don't want a video game to behave the same way every time we play!

Generating Random Numbers with Uniform Distribution

One simple kind of random number is one that takes on a real value that is *uniformly distributed* within some bounds.

You can get such numbers in R using the `runif` function. It takes as arguments the number of random numbers to generate, the low bound, and the high bound. We'll try generating one at a time here:

```
> runif(1,0,10)     # one random number in (0,10)
[1] 3.195956
> runif(1,0,10)     # another one, not the same
[1] 5.551191
> runif(1,0,10)     # ... and another
[1] 1.165307
> runif(1,100,200)  # one from a different range
[1] 182.0236
```

The random numbers generated are supposed to be *independent* — eg, which one we get the second time is unrelated to what the first one was.

R's Random Numbers Aren't Really Random

Computers are carefully designed to *not* behave randomly.

Some computers have special devices for producing random numbers that are really random. This is useful for cryptography (you want a really random key for your code, so nobody else can guess it).

But for most purposes we don't actually want real random numbers. They're too hard to generate, and if we use them, we can't reproduce our results another day.

For example: Imagine that after running your program for a long time, it stops with an error message, indicating it has a bug. You think you've now fixed the bug. But how do you verify that you've really fixed it if you can't reproduce the run that led to the error?

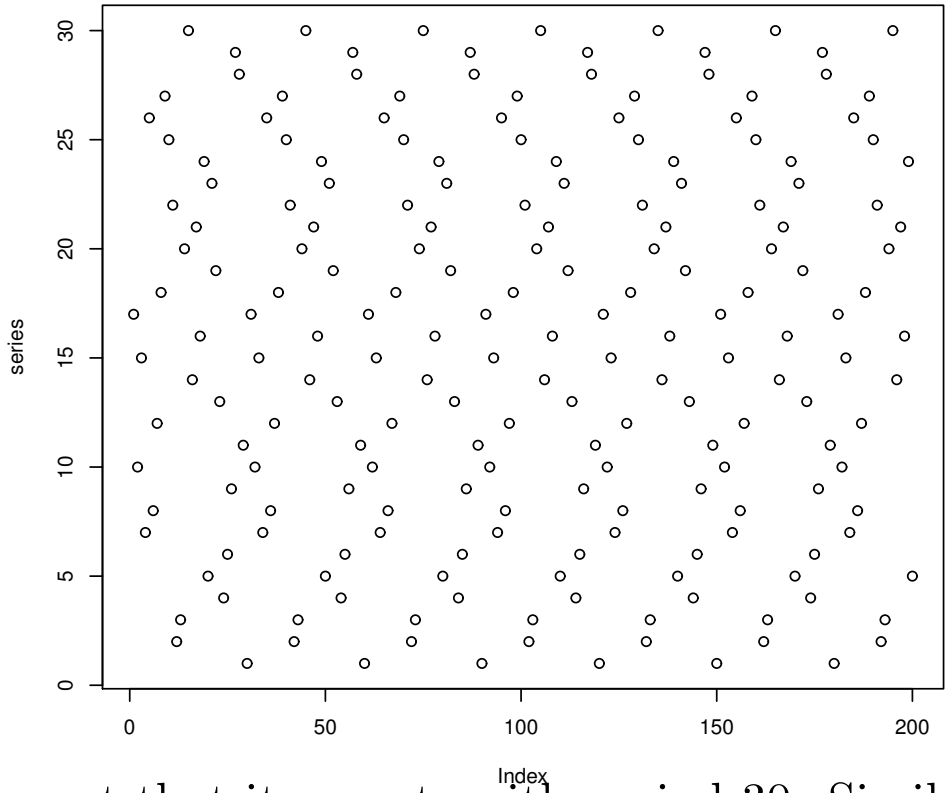
So most computer "random" numbers are really "pseudo-random" — numbers that *look random* for most purposes, but are actually generated by an algorithm that isn't random at all, so if it is run again, it will generate exactly the same numbers.

An Example of a Pseudo-Random Generator

Here's one simple way to generate a series of pseudo-random numbers, uniformly distributed over the integers 1, 2, . . . , 30.

```
> nxt <- 1; series <- c()
> for (i in 1:200) { nxt <- (nxt * 17) %% 31; series <- c(series,nxt) }
```

Here's a plot of the resulting series:



It looks random, except that it repeats with period 30. Similar generators can have much longer periods, however.

Setting the Random Seed

R uses a more sophisticated pseudo-random generator, but it also is deterministic, and will reproduce the same sequence if restarted with the same "seed".

For example:

```
> set.seed(123)
> runif(1)
[1] 0.2875775
> runif(1)
[1] 0.7883051
> runif(1)
[1] 0.4089769
> set.seed(123)
> runif(1)
[1] 0.2875775
> runif(1)
[1] 0.7883051
> runif(1)
[1] 0.4089769
```

For serious work, you should set the seed, so you'll be able to reproduce your results.

The `sample` function

The call `sample(n)` will generate a random permutation of the integers from 1 to `n`, as illustrated below:

```
> set.seed(1)
> sample(10)
 [1]  3  4  5  7  2  8  9  6 10  1
> sample(10)
 [1]  3  2  6 10  5  7  8  4  1  9
> sample(10)
 [1] 10  2  6  1  9  8  7  5  3  4
```

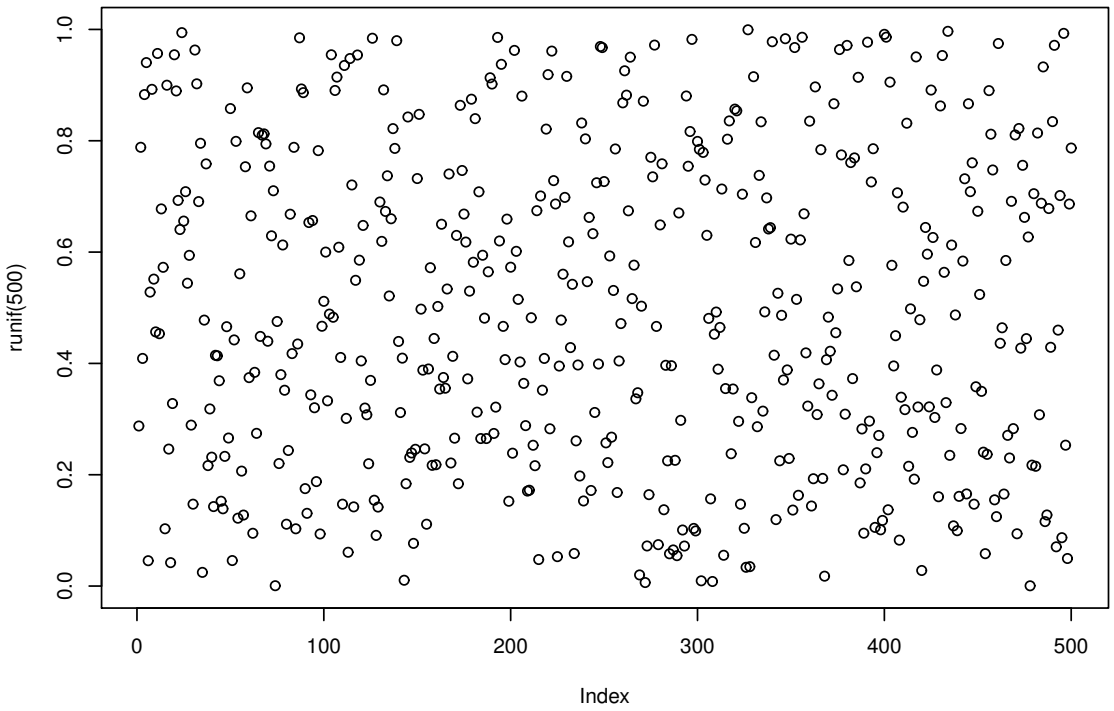
With other kinds of arguments, `sample` can do other things as well, including sampling with replacement.

Generating Random Vectors

The `runif` function can generate a whole vector of random numbers at once. The first argument of `runif` is the number of random numbers to generate.

For instance, here we plot 500 random numbers uniformly distributed from 0 to 1, using the command

```
> plot(runif(500))
```

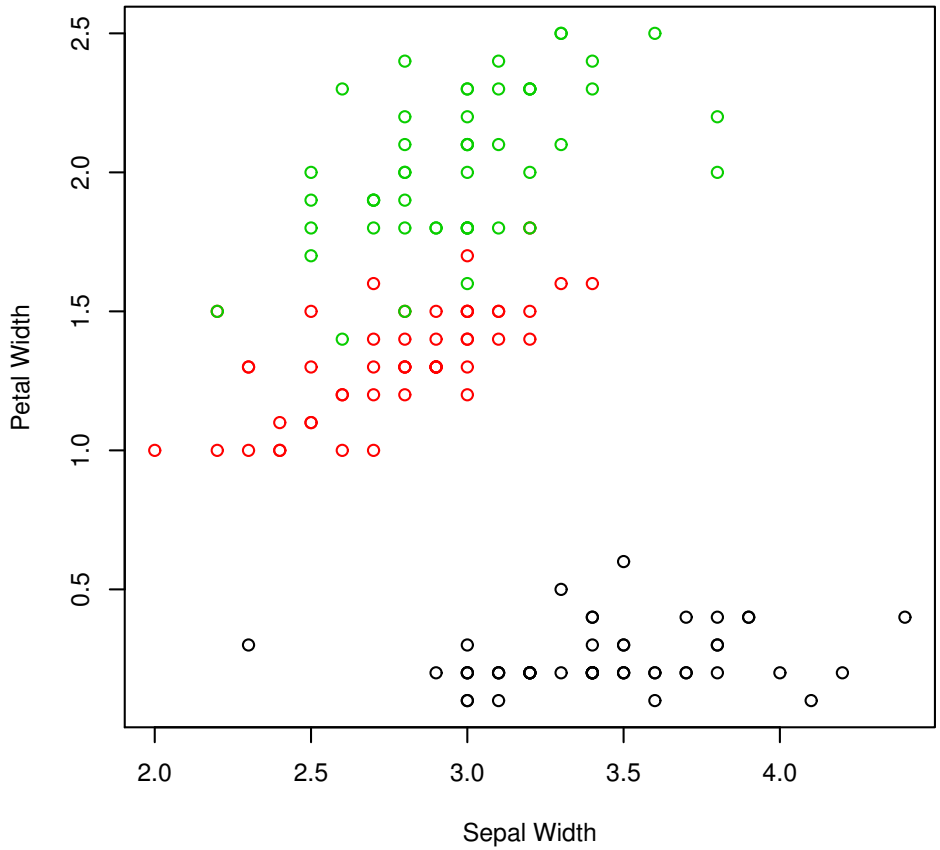


The Problem with Plotting Rounded Data Points

Recall the "iris" data set of width and length of petals and sepals in three species of Iris. It is stored in a special kind of list called a "data frame", which also looks sort-of like a matrix, which we'll talk more about later.

Here's a scatterplot of two of the variables (species marked by colour):

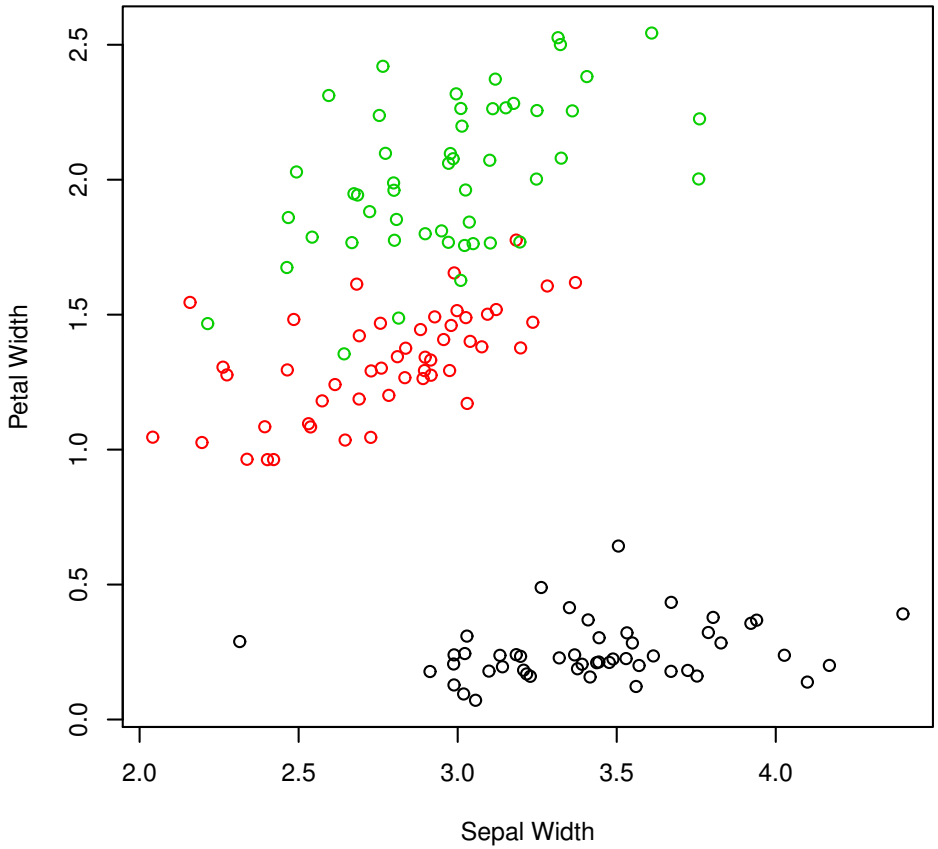
```
plot (iris$Sepal.Width, iris$Petal.Width, col=iris$Species,
      xlab="Sepal Width", ylab="Petal Width")
```



Solving the Problem with Random Jitter

Because the data is rounded to one decimal place, many of the dots in the scatterplot are on top of each other. To see all the data points, we can add random "jitter" to each data point before plotting:

```
plot (iris$Sepal.Width + runif(nrow(iris),-0.05,+0.05),
      iris$Petal.Width + runif(nrow(iris),-0.05,+0.05),
      col=iris$Species, xlab="Sepal Width", ylab="Petal Width")
```



Making Random Choices

Often, we want to make a random choice, with certain probabilities for doing certain things.

If we have a binary choice (to do or not do something), we can compare a random number that's uniform over $(0, 1)$ to the desired probability.

For example, at some point in a computer game, we might want to kill the player and end the game with probability 0.15. We can do it as follows:

```
if (runif(1) < 0.15) stop("You're dead. Game over!")
```

Why does this work?

Suppose instead we have a three-way choice – do A with probability 0.15, do B with probability 0.4, or do C with probability 0.45. (Note that these three probabilities add to one.)

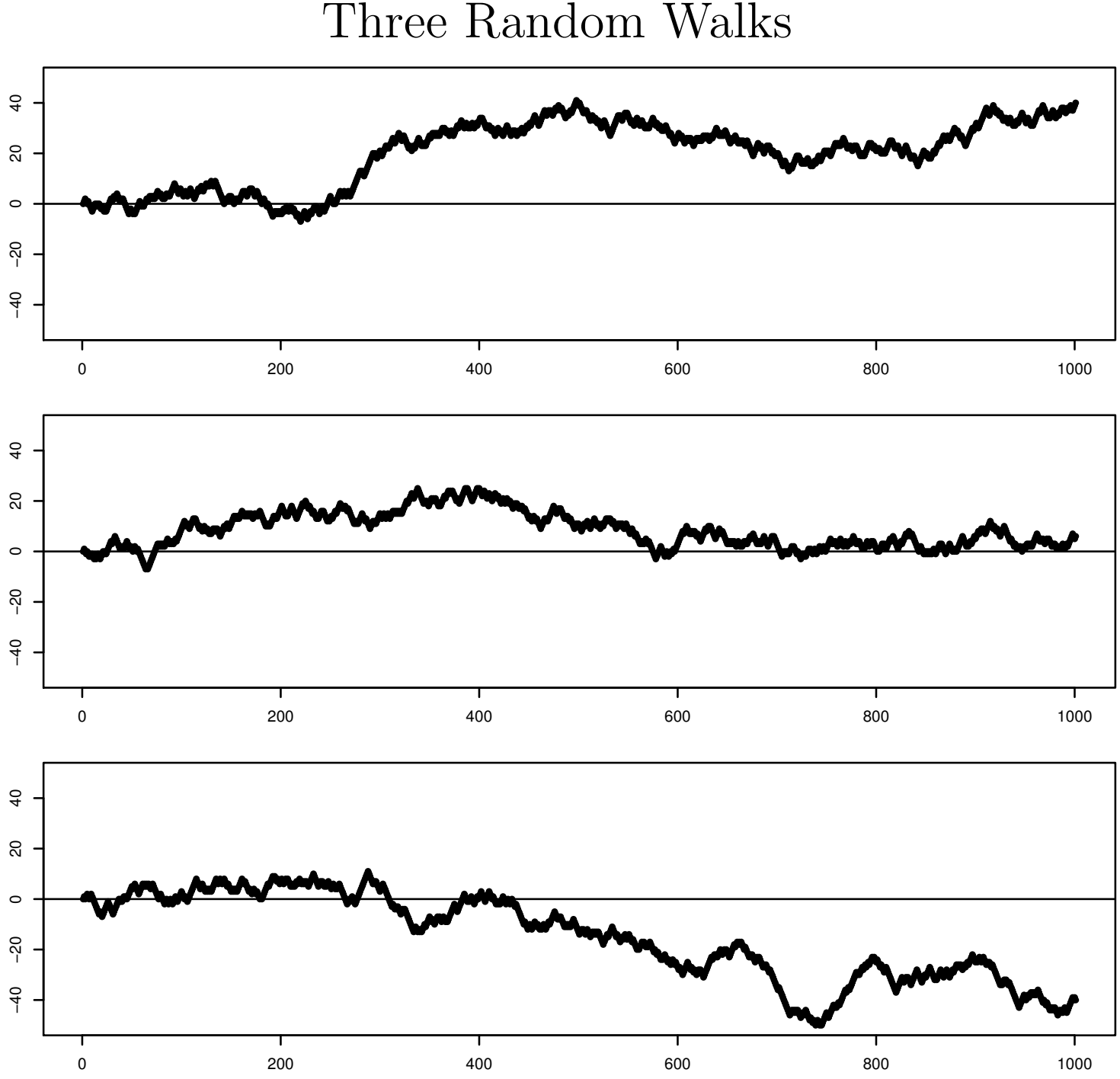
Could we generate one random number uniform over $(0, 1)$ and use it to make this choice?

Simulating a Random Walk

One well known "stochastic process" is a *random walk* on the integers, in which we start at 0, and at each time step thereafter we randomly go to the position one above or one below our current position, with probability 0.5 for either direction.

Here's an R function to simulate a random walk:

```
random_walk <- function (steps) {
    position <- numeric(steps+1)
    for (i in 1:steps) {
        if (runif(1) < 0.5)
            position[i+1] <- position[i] + 1
        else
            position[i+1] <- position[i] - 1
    }
    position
}
```


Three Random Walks
40
20
0
−20
−40
0
200
400
600
800
1000

Environments

An R *environment* is a collection of variables and their current values.

The *global* environment contains variables that are created when you assign to a name in a command typed at the R console (or as if typed in an R script).

For example, typing the command below creates (if it didn't exist already) a variable in the global environment named `fred`:

```
> fred <- 1+2
```

Calling a function creates a *local* environment used for just that call. Assignments inside the function create or change variables in that environment — below, the assignment to `fred` inside `f` changes `fred` in the local, not global, environment:

```
> f <- function (x) { fred <- 2*x; fred+1 }
> fred
[1] 3
> f(100)
[1] 201
> fred
[1] 3
```

Listing and Removing Variables

You can see what variables exist in the environment that is currently being used with the `ls` function, which returns a vector of strings with the names of variables.

You can remove a variable from the current environment with `rm`.

Here's an example (which assumes you haven't already defined other variables in the global environment):

```
> a <- 1
> b <- 2
> ls()
[1] "a" "b"
> rm(a)
> ls()
[1] "b"
> a
Error: object 'a' not found
```

Note: After `x <- "b"`, calling `rm(x)` removes variable `x`, *not* variable `b`.

Function Arguments in the Local Environment

When a function is called, all its arguments become variables in its local environment. Their values are what is was specified in the call of the function, or their default values if they were not specified.

We can see this by printing the result of `ls` inside a function:

```
> f <- function (x,y=100,z=1000) { print(ls()); x + y + z }
> f(7,z=10)
[1] "x" "y" "z"
[1] 117
```

If we create new variables by assignment, they also are in the local environment:

```
> g <- function (x,y=100,z=1000) { a <- x + y + z; print(ls()); a }
> g(7)
[1] "a" "x" "y" "z"
[1] 1107
```

The global environment isn't changed when local variables are created for arguments or by assignment. So after doing the above, in a new R session, we see

```
> ls()
[1] "f" "g"
```

Local and Global Variable References

When you reference a variable inside a function, it refers to the *local* variable of that name, if it exists, and if not, to the *global* variable of that name, if it exists.

Here's an example:

```
> f <- function (xyz,def) {
+     print (abc)  # refers to the global variable 'abc'
+     print (xyz)  # refers to the local variable (argument) 'xyz'
+     print (def)  # refers to the local variable (argument) 'def'
+     xyz + def + abc
+ }
>
> abc <- 1
> def <- 2
>
> f(200,3000)
[1] 1
[1] 200
[1] 3000
[1] 3201
```

Changing Local and Global Variables Inside a Function

Assigning a value to a name with `<-` (or with `=`) from inside a function creates or changes the *local* variable with that name. Assigning a value to a name with `<<-` creates or changes the *global* variable with that name. Here's an example:

```
> g <- function () {
+     x <- a     # creates a local variable 'x', with value from global 'a'
+     a <- 10    # creates a local variable 'a'; global 'a' is not affected
+     b <<- 300  # changes the global variable 'b'; doesn't create a local 'b'
+     a + b + x  # here, 'a' refers to the new local 'a', not the global 'a'
+ }
> g()
Error in g() : object 'a' not found
> a <- 100
> b <- 200
> g()
[1] 410
> a
[1] 100
> b
[1] 300
> x
Error: object 'x' not found
```

Assigning to Arguments Doesn't Change Them

Since assignments with `<-` inside a function change only the *local* environment, assigning to a function argument doesn't change what the caller passed.

For example:

```
> h <- function (x) { x[1] <- 0; sum(x) } # sum all but first element
> a <- c(3,4,1,7)
> h(a)
[1] 12
> a          # the global variable 'a' was not changed
[1] 3 4 1 7
> x <- c(10,20,30)
> h(x)
[1] 50
> x          # global 'x' unchanged - not the same as the local 'x'!
[1] 10 20 30
```

Exception: R has some "special" functions that do alter their arguments — for example, as we've seen, `rm(x)` actually removes `x`!

When and How to Use Local and Global Variables

When writing a function, you should try to

- Separate *what* the function does from *how* it does it, so someone using the function only needs to understand the "what".
- Make what the function does be easy to describe and understand.
- Make what the function does be general, so it will be useful in many contexts.

Functions should usually get input from their arguments, not global variables — they're then more generally useful, as it's easy to use different arguments in calls. Functions should usually not assign to global variables. Putting intermediate results in global variables makes "how" the function works be visible. Returning information in global variables makes it hard to use the function in a general way.

There are exceptions:

- If many functions all refer to the same data, having them all refer to a global `data` variable may be easier than passing a `data` argument to all of them.
- Assigning to a global variable can be a convenient way to keep track of overall counts of how often something happended (eg, number of errors of some sort).
- Assigning some intermediate result to a global variable may help when debugging a program (but take it out once the program is working).