Getting started with simulating data in R: some helpful functions and how to use them

Ariel Muldoon

August 28, 2018

Contents

Overview **2**

Generating random numbers **2**

`rnorm()` to generate random numbers from the normal distribution 2

Example of using the simulated numbers from `rnorm()` 5

`runif()` pulls from the uniform distribution 5

Example of using the simulated numbers from `runif()` 6

Discrete counts with `rpois()` 7

Example of using the simulated numbers from `rpois()` 7

Generate character vectors with `rep()` **9**

Using `letters` and `LETTERS` 9

Repeat each element of a vector with `each` 10

Repeat a whole vector with the `times` argument 10

Set the output vector length with the `length.out` argument 10

Repeat each element a different number of `times` 11

Combining `each` with `times` 11

Combining `each` with `length.out` 11

Creating datasets with quantiative and categorical variables **11**

Simulate data with no differences among two groups 12

Simulate data with a difference among groups 13

Multiple quantitative variables with groups 14

Repeatedly simulate data with `replicate()` **16**

Simple example of `replicate()` 16

An equivalent `for()` loop example 17

Using `replicate()` to repeatedly make a dataset 18

Overview

There are many reasons we might want to simulate data in R, and I find being able to simulate data to be incredibly useful in my day-to-day work. But how does someone get started simulating data?

Today I'm going to take a closer look at some of the R functions that are useful to get to know when simulating data. These functions are all from base R packages, not in add-on packages, so some of them may already familiar to you.

Here's what we'll do today:

1. Simulate quantitative variables via random number generation with `rnorm()`, `runif()` and `rpois()`.
2. Generate character variables that represent groups via `rep()`. We'll explore how to create character vectors with different repeating patterns.
3. Create data with both quantitative and categorical variables, making use of functions from the first two steps above.
4. Learn to use `replicate()` to repeat the data simulation process many times.

Generating random numbers

One way to generate numeric data is to pull random numbers from some distribution. This can be done via the functions for generating random deviates. These functions always start with `r` (for "random").

The basic distributions that I use the most for generating random numbers are the normal (`rnorm()`) and uniform (`runif()`) distributions. We'll look at those today, plus the Poisson (`rpois()`) distribution for generating discrete counts.

There are many other distributions available as part of the **stats** package (e.g., binomial, F, log normal, beta, exponential, Gamma) and, as you can imagine, even more available in add-on packages. I recently needed to generate data from the Tweedie distribution to test a modeling tool, which I could do via package **tweedie**.

The `r` functions for a chosen distribution all work basically the same way. We define how many random numbers we want to generate in the first argument (`n`) and then define the parameters for the distribution we want to draw from. This is easier to see with practice, so let's get started.

`rnorm()` to generate random numbers from the normal distribution

I use `rnorm()` a lot, sometimes with good reason and other times when I need some numbers and I really don't care too much about what they are.

There are three arguments to `rnorm()`. From the `Usage` section of the documentation:

```
rnorm(n, mean = 0, sd = 1)
```

The `n` argument is the number of observations we want to generate. The `mean` and `sd` arguments show what the default values are (note that `sd` is the *standard deviation*, not the variance). Not all `r` functions have defaults to the arguments like this.

To get 5 random numbers from a $Normal(0, 1)$ (aka the *standard* normal) distribution we can write code like:

```
rnorm(5)
```

```
[1]  1.5599938  1.1580437 -0.5546728 -1.1930674  0.8365917
```

There are a couple things about this code and the output to discuss.

First, the code got me 5 numbers, which is what I wanted. However, the code itself isn't particularly clear. What I might refer to as lazy coding on my part can look pretty mysterious to someone reading my code (or to my future self reading my code). Since I used the default values for `mean` and `sd`, it's not clear exactly what distribution I drew the numbers from.

Writing out arguments for clearer code

Here's clearer code to do the same thing, where I write out the mean and standard deviation arguments explicitly even though I'm using the default values. It is certainly not necessary to always be this careful, but I don't think I've run into a time when it was bad to have clear code.

```
rnorm(n = 5, mean = 0, sd = 1)
```

```
[1]  1.4364418  1.7637668 -1.4425682  0.1769415  0.5965349
```

Setting the random seed for reproducible random numbers

Second, if we run this code again we'll get different numbers. To get reproducible random numbers we need to *set the seed* via `set.seed()`.

Making sure someone else will be able to exactly reproduce your results when running the same code can be desirable in teaching. It is also is useful when making an example dataset to demonstrate a coding issue, like if you were asking a code question on Stack Overflow.

I also will set the seed when I'm making a function for simulations and I want to make sure it works correctly. Otherwise in most simulations we don't actually want or need to set the seed.

If we set the seed prior to running `rnorm()`, we can reproduce the values we generate.

```
set.seed(16)
rnorm(n = 5, mean = 0, sd = 1)
```

```
[1]  0.4764134 -0.1253800  1.0962162 -1.4442290  1.1478293
```

If we set the seed back to the same number and run the code again, we get the same values.

```
set.seed(16)
rnorm(n = 5, mean = 0, sd = 1)
```

```
[1]  0.4764134 -0.1253800  1.0962162 -1.4442290  1.1478293
```

Change parameters in `rnorm()`

For getting a quick set of numbers it's easy to use the default parameter values in `rnorm()` but we can certainly change the values to something else. For example, when I'm exploring long-run behavior of variance estimated from linear models I will want to vary the standard deviation values.

The `sd` argument shows the *standard deviation* of the normal distribution. So drawing from a $Normal(0, 4)$ can be done by setting `sd` to 2.

```
rnorm(n = 5, mean = 0, sd = 2)
```

```
[1] -0.9368241 -2.0119012  0.1271254  2.0499452  1.1462840
```

I've seen others change the mean and standard deviation to create a variable that is within some specific range, as well. For example, if the mean is large and the standard deviation small in relation to the mean we can generate strictly positive numbers. (I usually use `runif()` for this, which we'll see below.)

```
rnorm(n = 5, mean = 50, sd = 20)
```

```
[1] 86.94364 52.23867 35.07925 83.16427 64.43441
```

Using vectors of values for the parameter arguments

We can pull random numbers from a normal distribution with distinct parameters if we use a vector for the parameter arguments. For example, this could be useful for simulating data with different group means but the same variance. We might want to use something like this if we were making data that we would analyze using an ANOVA.

I'll keep the standard deviation at 1 but will draw data from three distribution centered at three different locations: one at 0, one at 5, and one at 20. I request 10 total draws by changing `n` to 10.

Note the repeating pattern: the function iteratively draws one value from each distribution until the total number requested is reached. This can lead to imbalance in the sample size per distribution.

```
rnorm(n = 10, mean = c(0, 5, 20), sd = 1)
```

```
 [1] -1.6630805  5.5759095 20.4727601 -0.5427317  6.1276871 18.3522024
 [7] -0.3141739  4.8173184 21.4704785 -0.8658988
```

A vector can also be passed to `sd`. Here both the means and standard deviations vary among the three distributions used to generate values.

```
rnorm(n = 10, mean = c(0, 5, 20), sd = c(1, 5, 20) )
```

```
 [1]  1.5274670 10.2708903 40.6014202  0.8401609  6.0848235  6.5494885
 [7]  0.1325985  4.6453633  1.1460906 -1.0220310
```

Things are different for the `n` argument. If a vector is passed to `n`, the *length* of that vector is taken to be the number required (see `Arguments` section of documentation for details).

Here's an example. Since the vector for `n` is length 3, we only get 3 values.

```
rnorm(n = c(2, 10, 10), mean = c(0, 5, 20), sd = c(1, 5, 20) )
```

```
[1]  0.2805551  7.7239168 22.6173950
```

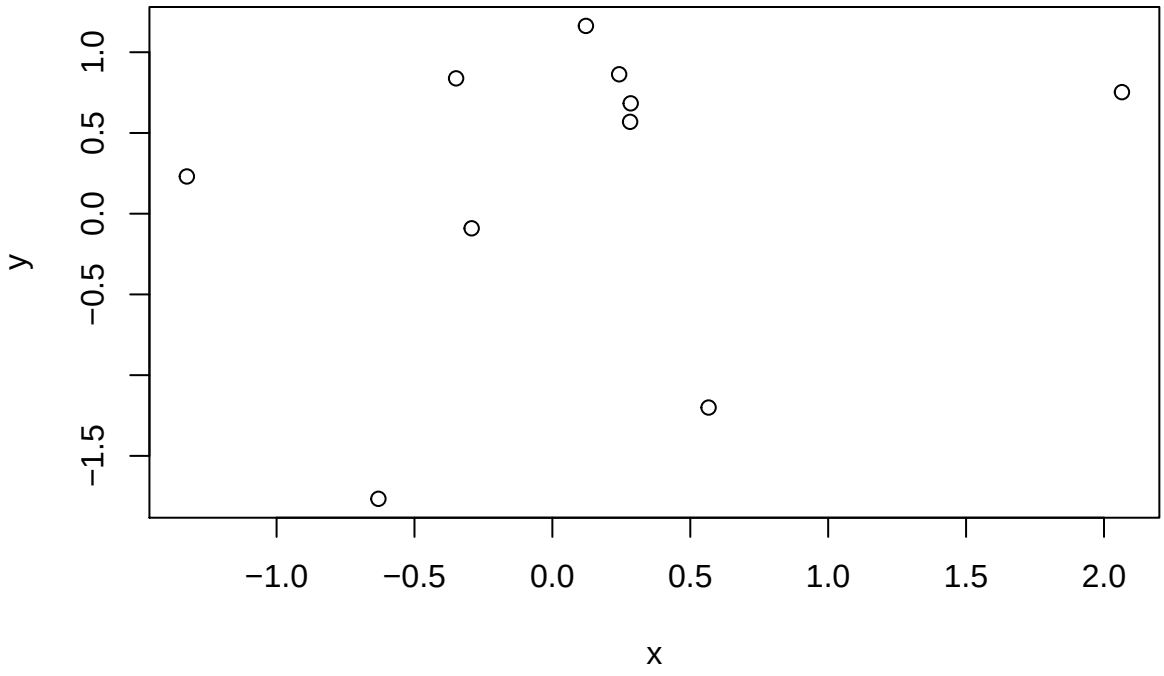
Example of using the simulated numbers from `rnorm()`

Up to this point we've printed the results of each simulation. In reality we'd want to save our vectors as objects in R to use them for some further task.

For example, maybe we want to simulate two unrelated variables and then look to see how correlated they appear to be. This can be a fun exercise to demonstrate how variables can appear to be related by chance even when we know they are not, especially at small sample sizes.

Let's generate two quantitative vectors of length 10, which I'll name `x` and `y`, and plot the results. I'm using the defaults for `mean` and `sd`.

```
x = rnorm(n = 10, mean = 0, sd = 1)
y = rnorm(n = 10, mean = 0, sd = 1)
plot(y ~ x)
```



`runif()` pulls from the uniform distribution

Pulling random numbers from other distributions is extremely similar to using `rnorm()`, so we'll go through them more quickly.

I've started using `runif()` pretty regularly, especially when I want to easily generate numbers that are strictly positive but continuous. The uniform distribution is a continuous distribution, with numbers uniformly distributed between some minimum and maximum.

From `Usage` we can see that by default we pull random numbers between 0 and 1. The first argument, as with all of these `r` functions, is the number of deviates we want to randomly generate:

runif(n, min = 0, max = 1)

Let's generate 5 numbers between 0 and 1.

```
runif(n = 5, min = 0, max = 1)
```

```
[1] 0.9994220 0.9432766 0.2496042 0.6482484 0.1125788
```

What if we want to generate 5 numbers between 50 and 100? We can change the parameter values.

```
runif(n = 5, min = 50, max = 100)
```

```
[1] 81.56680 66.91604 82.70027 64.29244 54.17023
```

Example of using the simulated numbers from `runif()`

One situation where I've found `runif()` handy is when I wanted to demonstrate the effect of the relative size of the variable values on the size of the estimated coefficient in a regression. For example, the size of the coefficient measured in kilometers is smaller than if that variable was converted into meters.

Let's generate some data with the response variable (`y`) pulled from a standard normal distribution and then an explanatory variable with values between 1 and 2. The two variables are unrelated.

You see I'm still writing out my argument names for clarity, but you may get a sense how easy it would be to start cutting corners to avoid the extra typing.

```
set.seed(16)
y = rnorm(n = 100, mean = 0, sd = 1)
x1 = runif(n = 100, min = 1, max = 2)
head(x1)
```

```
[1] 1.957004 1.082791 1.710816 1.326998 1.995723 1.449522
```

Now simulate a second explanatory variable with values between 200 and 300. This variable is also unrelated to the other two.

```
x2 = runif(n = 100, min = 200, max = 300)
head(x2)
```

```
[1] 220.0617 263.4875 209.6036 245.3125 265.1869 257.4817
```

We can fit a multiple regression linear model via `lm()`. The coefficient for the second variable, with a larger relative size, is generally going to be smaller than the first since the change in `y` for a "1-unit increase" in `x` depends on the units of `x`.

```
lm(y ~ x1 + x2)
```

```
Call:
lm(formula = y ~ x1 + x2)

Coefficients:
(Intercept)           x1           x2
   0.380887     0.104941    -0.001908
```

Discrete counts with `rpois()`

Let's look at one last function for generating random numbers, this time for generating discrete integers (including 0) from a Poisson distribution with `rpois()`.

I use `rpois()` for generating counts for exploring generalized linear models. I've also found this function useful in gaining a better understanding of the shape of Poisson distributions with different means.

The Poisson distribution is a single parameter distribution. The function looks like:

    rpois(n, lambda)

The single parameter, `lambda`, is the mean. It has no default setting so must always be defined by the user.

Let's generate five values from a Poisson distribution with a mean of 2.5. Note that *mean* of the Poisson distribution can be any non-negative value (i.e., it doesn't have to be an integer) even though the observed values will be discrete integers.

```
rpois(n = 5, lambda = 2.5)
```

```
[1] 2 1 4 1 2
```

Example of using the simulated numbers from `rpois()`

Let's explore the Poisson distribution a little more, seeing how the distribution looks when changing the mean. Being able to look at how the Poisson distribution changes with the mean via simulation helped me understand the distribution better, including why it so often does a poor job modeling ecological count data.

We'll draw 100 values from a Poisson distribution with a mean of 5. We'll name this vector `y` and take a look at a summary of those values.

```
y = rpois(100, lambda = 5)
```

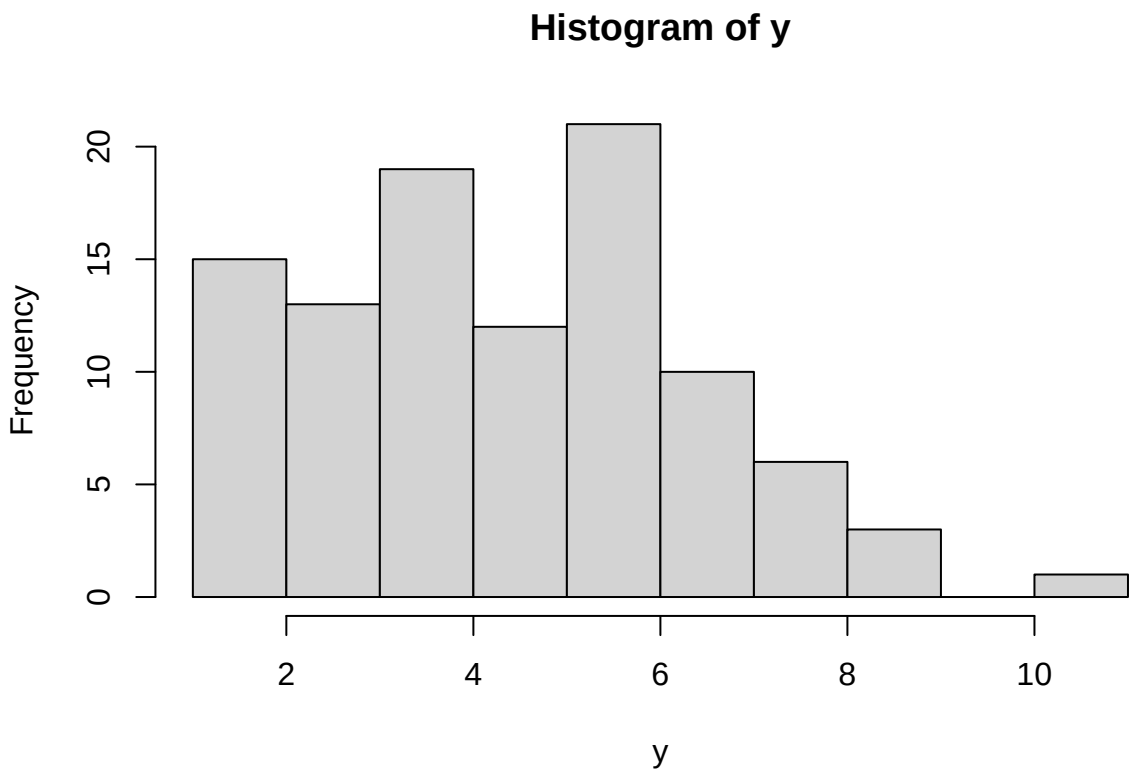
The vector of values we simulated fall between 1 and 14.

```
summary(y)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   1.00    3.00    5.00    4.83    6.00   11.00
```

There is mild right-skew when we draw a histogram of the values.

```
hist(y)
```



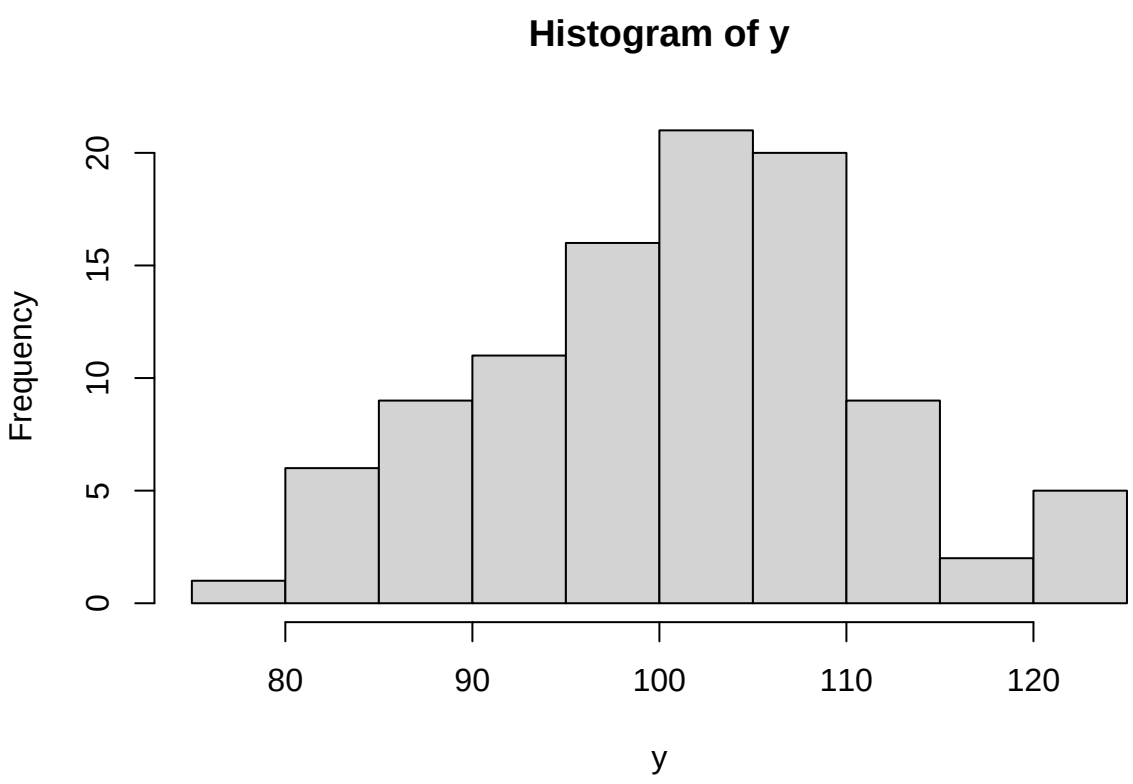
Let's do the same thing for a Poisson distribution with a mean of 100. The range of values is pretty narrow; there are no values even remotely close to 0.

```
y = rpois(100, lambda = 100)
summary(y)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  76.00   94.75  102.00  101.31  108.00  124.00
```

And the distribution is pretty symmetric compared to the distribution with the smaller mean.

```
hist(y)
```



An alternative to the Poisson distribution for discrete integers is the negative binomial distribution. Packages **MASS** has a function called `rnegbin()` for random number generation from the negative binomial distribution.

Generate character vectors with `rep()`

Quantitative variables are great, but in simulations we're often going to need categorical variables, as well.

In my own work these are usually sort of "grouping" or "treatment" variable. This means I need to have repeated observations of each character value. The `rep()` function is one way to avoid having to write out an entire vector manually. It's for *replicating elements of vectors and lists.*

Using `letters` and `LETTERS`

The first argument of `rep()` is the vector to be repeated. One option is to write out the character vector you want to repeat. You can also get a simple character vector through the use of `letters` or `LETTERS`. These are *built in constants* in R. `letters` is the 26 lowercase letters of the Roman alphabet and `LETTERS` is the 26 uppercase letters.

Letters can be pulled out via the extract brackets (`[`). I use these built-in constants for pure convenience when I need to make a basic categorical vector and it doesn't matter what form those categories take. I find it more straightforward to type out the word and brackets than a vector of characters (complete with all those pesky quotes and such).

Here are the first two `letters`.

```
letters[1:2]
```

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

And the last 17 `LETTERS`.

```
LETTERS[10:26]
```

```
 [1] "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
```

Repeat each element of a vector with `each`

There are three arguments that help us repeat the values in the vector in `rep()` with different patterns: `each`, `times`, and `length.out`. These can be used individually or in combination.

With `each` we repeat each unique character in the vector the defined number of times. The replication is done "elementwise", so the repeats of each unique character are all in a row.

Let's repeat two characters three times each. The resulting vector is 6 observations long.

This is an example of how I might make a grouping variable for simulating data to be used in a two-sample analysis.

```
rep(letters[1:2], each = 3)
```

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

Repeat a whole vector with the `times` argument

The `times` argument can be used when we want to repeat the whole vector rather than repeating it elementwise.

We'll make a two-group variable again, but this time we'll change the repeating pattern of the values in the variable.

```
rep(letters[1:2], times = 3)
```

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

Set the output vector length with the `length.out` argument

The `length.out` argument has `rep()` repeat the whole vector. However, it repeats the vector only until the defined length is reached. Using `length.out` is another way to get unbalanced groups.

Rather than defining the number of repeats like we did with `each` and `times` we define the length of the output vector.

Here we'll make a two-group variable of length 5. This means the second group will have one less value than the first.

```
rep(letters[1:2], length.out = 5)
```

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

Repeat each element a different number of `times`

Unlike `each` and `length.out`, we can use `times` with a vector of values. This allows us to repeat each element of the character vector a different number of times. This is one way to simulate unbalanced groups. Using `times` with a vector repeats each element like `each` does, which can make it harder to remember which argument does what.

Let's repeat the first element twice and the second four times.

```
rep(letters[1:2], times = c(2, 4) )
```

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

Combining `each` with `times`

As your simulation situation get more complicated, you may need more complicated patterns for your categorical variable. The `each` argument can be combined with `times` to first repeat each value elementwise (via `each`) and then repeat that whole pattern (via `times`).

When using `times` this way it will only take a single value and not a vector.

Let's repeat each value twice, 3 times.

```
rep(letters[1:2], each = 2, times = 3)
```

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

Combining `each` with `length.out`

Similarly we can use `each` with `length.out`. This can lead to some imbalance.

Here we'll repeat the two values twice each but with a total final vector length of 7.

```
rep(letters[1:2], each = 2, length.out = 7)
```

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

Note you can't use `length.out` and `times` together (if you try, `length.out` will be given priority and `times` ignored).

Creating datasets with quantiative and categorical variables

We now have some tools for creating quantitative data as well as categorical. Which means it's time to make some datasets! We'll create several simple ones to get the general idea.

Simulate data with no differences among two groups

Let's start by simulating data that we would use in a simple two-sample analysis with no difference between groups. We'll make a total of 6 observations, three in each group.

We'll be using the tools we reviewed above but will now name the output and combine them into a data.frame. This last step isn't always necessary, but does help keep things organized in certain types of simulations.

First we'll make separate vectors for the continuous and categorical data and then bind them together via `data.frame()`.

Notice there is no need to use `cbind()` here, which is commonly done by R beginners (I know I did!). Instead we can use `data.frame()` directly.

```
group = rep(letters[1:2], each = 3)
response = rnorm(n = 6, mean = 0, sd = 1)
data.frame(group,
           response)
```

```
  group   response
1     a  0.4933614
2     a  0.5234101
3     a  1.2365975
4     b  0.3563153
5     b  0.5748968
6     b -0.4222890
```

When I make a data.frame like this I prefer to make my vectors and the data.frame simultaneously to avoid having a lot of variables cluttering up my R Environment.

I often teach/blog with all the steps clearly delineated as I think it's easier when you are starting out, so (as always) use the method that works for you.

```
data.frame(group = rep(letters[1:2], each = 3),
           response = rnorm(n = 6, mean = 0, sd = 1) )
```

```
  group   response
1     a  0.4024228
2     a  0.9585800
3     a -1.8763844
4     b -0.2115171
5     b  1.4374372
6     b  0.3855285
```

Now let's add another categorical variable to this dataset.

Say we are in a situation involving two factors, not one. We have a single observations for every combination of the two factors (i.e., the two factors are *crossed*).

The second factor, which we'll call `factor`, will take on the values "C", "D", and "E".

```
LETTERS[3:5]
```

```
[1] "C" "D" "E"
```

We need to repeat the values in a way that every combination of `group` and `factor` is present in the dataset at one time.

Remember the `group` factor is repeated elementwise.

```
rep(letters[1:2], each = 3)
```

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

We need to repeat the three values twice. But what argument do we use in `rep()` to do so?

```
rep(LETTERS[3:5], ?)
```

Does `each` work?

```
rep(LETTERS[3:5], each = 2)
```

```
[1] "C" "C" "D" "D" "E" "E"
```

No, if we use `each` then each element is repeated twice and some of the combinations of `group` and `factor` are missing.

This is a job for the `times` or `length.out` arguments, so the whole vector is repeated. We can repeat the whole vector twice using `times`, or use `length.out = 6`. I do the former.

In the result below we can see every combination of the two factors is present once.

```
data.frame(group = rep(letters[1:2], each = 3),
           factor = rep(LETTERS[3:5], times = 2),
           response = rnorm(n = 6, mean = 0, sd = 1) )
```

```
  group factor   response
1     a      C  0.4255576
2     a      D  0.2903586
3     a      E -0.3638877
4     b      C  1.9778117
5     b      D  1.0869069
6     b      E -0.5869200
```

Simulate data with a difference among groups

The dataset above is one with "no difference" among groups. What if the means were different between groups? Let's make two groups of three observations where the mean of one group is 5 and the other is 10. The two groups have a shared variance (and so standard deviation) of 1.

Remembering how `rnorm()` works with a vector of means is key here. The function draws iteratively from each distribution.

```
response = rnorm(n = 6, mean = c(5, 10), sd = 1)
response
```

```
[1]  4.413753 12.484499  4.740506 10.273258  5.369074 10.024199
```

How do we get the `group` pattern correct?

```
group = rep(letters[1:2], ?)
```

We need to repeat the whole vector three times instead of elementwise.

To get the groups in the correct order we need to use `times` or `length.out` in `rep()`. With `length.out` we define the output length of the vector, which is 6. Alternatively we could use `times = 3` to repeat the whole vector 3 times.

```
group = rep(letters[1:2], length.out = 6)
group
```

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

These can then be combined into a data.frame. Working out this process is another reason why sometimes we build each vector separately prior to combining together.

```
data.frame(group,
           response)
```

```
  group  response
1     a  4.413753
2     b 12.484499
3     a  4.740506
4     b 10.273258
5     a  5.369074
6     b 10.024199
```

Multiple quantitative variables with groups

For our last dataset we'll have two groups, with 10 observations per group.

```
rep(LETTERS[3:4], each = 10)
```

```
 [1] "C" "C" "C" "C" "C" "C" "C" "C" "C" "C" "D" "D" "D" "D" "D" "D" "D" "D" "D"
[20] "D"
```

Let's make a dataset that has two quantitative variables, unrelated to both each other and the groups. One variable ranges from 10 and 15 and one from 100 and 150.

How many observations should we draw from each uniform distribution?

```
runif(n = ?, min = 10, max = 15)
```

We had 2 groups with 10 observations each and 2*10 = 20. So we need to use `n = 20` in `runif()`.

Here is the dataset made in a single step.

```r
data.frame(group = rep(LETTERS[3:4], each = 10),
           x = runif(n = 20, min = 10, max = 15),
           y = runif(n = 20, min = 100, max = 150))
```

```
   group        x        y
1      C 13.20331 126.7004
2      C 13.91440 137.0772
3      C 12.72031 134.8689
4      C 14.27637 122.7582
5      C 11.72933 118.0189
6      C 14.59640 108.4544
7      C 12.81629 142.1792
8      C 13.45864 104.0355
9      C 13.94198 107.0199
10     C 12.15106 145.4622
11     D 12.01762 117.3381
12     D 13.66006 121.2900
13     D 14.78914 145.2424
14     D 11.70157 120.1363
15     D 13.25180 139.8288
16     D 10.76914 106.6282
17     D 13.97263 147.6383
18     D 14.91621 112.6612
19     D 13.53270 104.8965
20     D 14.40677 119.5231
```

What happens if we get this wrong? If we're lucky we get an error.

```r
data.frame(group = rep(LETTERS[3:4], each = 10),
           x = runif(n = 15, min = 10, max = 15),
           y = runif(n = 15, min = 100, max = 150))
```

```
Error in data.frame(group = rep(LETTERS[3:4], each = 10), x = runif(n =
15, : arguments imply differing number of rows: 20, 15```

But if we get things wrong and the number we use goes into the number we need evenly, R will *recycle* 

This is a hard mistake to catch.  If you look carefully through the output below you can see that the c


```r
data.frame(group = rep(LETTERS[3:4], each = 10),
           x = runif(n = 10, min = 10, max = 15),
           y = runif(n = 10, min = 100, max = 150))
```

```
   group        x        y
1      C 12.28493 108.3455
2      C 13.84490 114.8247
3      C 12.42386 105.1358
4      C 10.08725 125.1979
5      C 10.83277 129.6310
6      C 10.96766 129.4584
```
```

```
7      C 11.51180 149.2819
8      C 13.48253 139.2530
9      C 11.64337 119.6663
10     C 12.88603 119.8368
11     D 12.28493 108.3455
12     D 13.84490 114.8247
13     D 12.42386 105.1358
14     D 10.08725 125.1979
15     D 10.83277 129.6310
16     D 10.96766 129.4584
17     D 11.51180 149.2819
18     D 13.48253 139.2530
19     D 11.64337 119.6663
20     D 12.88603 119.8368
```

Repeatedly simulate data with `replicate()`

The `replicate()` function is a real workhorse when making repeated simulations. It is a member of the *apply* family in R, and is specifically made (per the documentation) for the *repeated evaluation of an expression (which will usually involve random number generation)*.

We want to repeatedly simulate data that involves random number generation, so that sounds like a useful tool.

The `replicate()` function takes three arguments:

- `n`, which is the number of replications to perform. This is to set the number of repeated runs we want.
- `expr`, the expression that should be run repeatedly. This is often a function.
- `simplify`, which controls the type of output the results of `expr` are saved into. Use `simplify = FALSE` to get output saved into a list instead of in an array.

Simple example of `replicate()`

Let's say we want to simulate some values from a normal distribution, which we can do using the `rnorm()` function as above. But now instead of drawing some number of values from a distribution one time we want to do it many times. This could be something we'd do when demonstrating the central limit theorem, for example.

Doing the random number generation many times is where `replicate()` comes in. It allows us to run the function in `expr` exactly `n` times.

Here I'll generate 5 values from a standard normal distribution three times. Notice the addition of `simplify = FALSE` to get a list as output.

The output below is a list of three vectors. Each vector is from a unique run of the function, so contains five random numbers drawn from the normal distribution with a mean of 0 and standard deviation of 1.

```
set.seed(16)
replicate(n = 3,
          expr = rnorm(n = 5, mean = 0, sd = 1),
          simplify = FALSE )
```

```
[[1]]
[1]  0.4764134 -0.1253800  1.0962162 -1.4442290  1.1478293

[[2]]
[1] -0.46841204 -1.00595059  0.06356268  1.02497260  0.57314202

[[3]]
[1]  1.8471821  0.1119334 -0.7460373  1.6582137  0.7217206
```

Note if I don't use `simplify = FALSE` I will get a matrix of values instead of a list. Each column in the matrix is the output from one run of the function.

In this case there will be three columns in the output, one for each run, and 5 rows. This can be a useful output type for some simulations. I focus on list output throughout the rest of this post only because that's what I have been using recently for simulations.

```
set.seed(16)
replicate(n = 3,
          expr = rnorm(n = 5, mean = 0, sd = 1) )
```

```
           [,1]        [,2]       [,3]
[1,]  0.4764134 -0.46841204  1.8471821
[2,] -0.1253800 -1.00595059  0.1119334
[3,]  1.0962162  0.06356268 -0.7460373
[4,] -1.4442290  1.02497260  1.6582137
[5,]  1.1478293  0.57314202  0.7217206
```

An equivalent `for()` loop example

A `for()` loop can be used in place of `replicate()` for simulations. With time and practice I've found `replicate()` to be much more convenient in terms of writing the code. However, in my experience some folks find `for()` loops intuitive when they are starting out in R. I think it's because `for()` loops are more explicit on the looping process: the user can see the values that `i` takes and the output for each `i` iteration is saved into the output object because the code is written out explicitly.

In my example I'll save the output of each iteration of the loop into a list called `list1`. I initialize this as an empty list prior to starting the loop. To match what I did with `replicate()` I do three iterations of the loop (`i in 1:3`), drawing 5 values via `rnorm()` each time.

The result is identical to my `replicate()` code above. It took a little more code to do it but the process is very clear since it is explicitly written out.

```
set.seed(16)
list1 = list() # Make an empty list to save output in
for (i in 1:3) { # Indicate number of iterations with "i"
    list1[[i]] = rnorm(n = 5, mean = 0, sd = 1) # Save output in list for each iteration
}
list1
```

```
[[1]]
[1]  0.4764134 -0.1253800  1.0962162 -1.4442290  1.1478293

[[2]]
```

```
[1] -0.46841204 -1.00595059  0.06356268  1.02497260  0.57314202

[[3]]
[1]  1.8471821  0.1119334 -0.7460373  1.6582137  0.7217206
```

Using `replicate()` to repeatedly make a dataset

Earlier we were making datasets with random numbers and some grouping variables. Our code looked like

```
data.frame(group = rep(letters[1:2], each = 3),
           response = rnorm(n = 6, mean = 0, sd = 1) )
```

```
  group   response
1     a -1.6630805
2     a  0.5759095
3     a  0.4727601
4     b -0.5427317
5     b  1.1276871
6     b -1.6477976
```

We could put this process as the `expr` argument in `replicate()` to get many simulated datasets. I would do something like this if I wanted to compare the long-run performance of two different statistical tools using the exact same random datasets.

I'll replicate things 3 times again to easily see the output. I still use `simplify = FALSE` to get things into a list.

```
simlist = replicate(n = 3,
          expr = data.frame(group = rep(letters[1:2], each = 3),
                            response = rnorm(n = 6, mean = 0, sd = 1) ),
          simplify = FALSE)
```

We can see this result is a list of three data.frames.

```
str(simlist)
```

```
List of 3
 $ :'data.frame':   6 obs. of  2 variables:
  ..$ group   : chr [1:6] "a" "a" "a" "b" ...
  ..$ response: num [1:6] -0.314 -0.183 1.47 -0.866 1.527 ...
 $ :'data.frame':   6 obs. of  2 variables:
  ..$ group   : chr [1:6] "a" "a" "a" "b" ...
  ..$ response: num [1:6] 1.03 0.84 0.217 -0.673 0.133 ...
 $ :'data.frame':   6 obs. of  2 variables:
  ..$ group   : chr [1:6] "a" "a" "a" "b" ...
  ..$ response: num [1:6] -0.943 -1.022 0.281 0.545 0.131 ...
```

Here is the first one.

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

```
  group   response
1     a -0.3141739
2     a -0.1826816
3     a  1.4704785
4     b -0.8658988
5     b  1.5274670
6     b  1.0541781
```

What's the next step?

I'm ending here, but there's still more to learn about simulations. For a simulation to explore long-run behavior, some process is going to be repeated many times. We did this via `replicate()`. The next step would be to extract whatever results are of interest. This latter process is often going to involve some sort of looping.

By saving our generated variables or data.frames into a list we've made it so we can loop via list looping functions like `lapply()` or `purrr::map()`. The family of *map* functions are newer and have a lot of convenient output types that make them pretty useful. If you want to see how that might look for a simulations, you can see a few examples in my blog post A closer look at replicate() and purrr::map() for simulations.

Happy simulating!