Lab 2 Supplement: R Reference Guide

Statistics 139 (Special thanks: Julie Vu)

September 9, 2022

Random Sampling with `sample()`

The `sample()` function has the generic structure

```
sample(x, size = , replace = FALSE, prob = NULL)
```

where the `prob` argument allows for the probability of sampling each element in `x` to be specified as a vector. When `prob` is omitted, the function will sample each element with equal probability.

The following code simulates the outcome of tossing a biased coin ten times, where the probability of a heads is 0.6; a heads is represented by `1` and a tails is represented by `0`. The first argument is the vector `(0, 1)`, and the `prob` argument is the vector `(0.6, 0.4)`; the order indicates that the first element (`0`) is to be sampled with probability 0.6 and the second element (`1`) is to be sampled with probability 0.4.

```
outcomes = sample(c(0, 1), size = 10, prob = c(0.4, 0.6), replace = TRUE)
outcomes
```

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

Pseudorandom Sampling

To be able to retrieve a sample that has been generated, it is necessary to associate the sample with a seed (i.e., a tag). Setting the seed allows for the same sample to be generated with a subsequent run of `sample()`.

The following code shows samples of size 10 drawn from the integers 1 through 15, either with replacement or without replacement. In the first run, it is possible to obtain a specific number more than once; this is not possible in the other runs. The first two runs are not associated with a seed. The second two are associated with the seed `2018`; running `sample()` with these specifications and seed will always result in the sample 6, 7, 1, 3, 15, 4, 11, 2, 14, 10. Note how the last two runs result in a different sample than the second run.

```
#completely random
sample(1:15, size = 10, replace = TRUE)
```

```
##  [1]  9 14 13 10 15  8 15  4 14  1
```

```
sample(1:15, size = 10, replace = FALSE)
```

```
##  [1] 15  6  9 13  7  5  2  3  1 12
```

```
#set.seed
set.seed(2018)
sample(1:15, size = 10, replace = FALSE)
```

```
##  [1] 15  7  3 12  2  5  9 13 14  4
```

```r
set.seed(2018)
sample(1:15, size = 10, replace = FALSE)
```

```
##  [1] 15  7  3 12  2  5  9 13 14  4
```

Using `sum()`

The `sum()` function is used in the lab to return the sum of the outcomes vector, where outcomes are recorded as either `0` or `1`. For simplicity, the function was used in its most basic form:

```r
sum(outcomes)        #number of 1's (heads)
```

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

```r
10 - sum(outcomes)   #number of 0's (tails)
```

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

It is often convenient to combine the `sum()` function with logical operators.

```r
sum(outcomes == 1)   #number of 1's (heads)
```

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

```r
sum(outcomes == 0)   #number of 0's (tails)
```

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

This provides more flexibility and can make the code easier to parse quickly, such as for cases where there are more than two outcomes, or when more than two outcomes are of interest. For example, the following application of `sum()` identifies the number of rolls (out of twenty) of a fair six-sided that are either 1 or greater than 4.

```r
#set the seed for a pseudo-random sample
set.seed(2018)

dice.rolls = sample(1:6, size = 20, replace = TRUE)
dice.rolls
```

```
##  [1] 3 4 5 2 5 1 3 4 2 4 3 3 6 1 1 6 5 3 1 3
```

```r
sum(dice.rolls == 1 | dice.rolls > 4)
```

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

Additionally, since logical operators work with string values, it is possible to use them with `sum()` to return the number of heads if heads is represented by, for example, `H`.

```r
#set the seed for a pseudo-random sample
set.seed(2018)

outcomes = sample(c("T", "H"), size = 10, prob = c(0.4, 0.6), replace = TRUE)
outcomes
```

```
##  [1] "H" "H" "H" "H" "H" "H" "T" "H" "T" "H"
```

```
sum(outcomes == "H")  #number of heads
```

```
## [1] 8
```

`for` Loops

A loop allows for a set of code to be repeated under a specific set of conditions; the `for` loop is one of the several types of loops available in R.

A `for` loop has the basic structure `for( counter ) { instructions }`. The loop below will calculate the squares of the integers from 1 through 5.

- Prior to running the loop, an empty vector `squares` is created to store the results of the loop. This step is referred to as initialization.
- The counter consists of the index variable that keeps track of each iteration of the loop; the index is typically a letter like `i`, `j`, or `k`, but can be any sequence such as `ii`. The index variable is conceptually similar to the index of summation k in sigma notation ($\sum_{k=1}^{n}$). In the example below, the counter can be read as "for every k in 1 through 5, repeat the following instructions..."
- The instructions are enclosed within the pair of curly braces. For each iteration, the value k^2 is to be stored in the k^{th} element of the vector `squares`. The empty vector `squares` was created prior to running the loop using `vector()`.
- So, for the first iteration, `R` sets $k = 1$, calculates 1^2, and fills in the first element of `squares` with the result. In the next iteration, $k = 2\ldots$ and this process repeats until $k = 5$ and 5^2 is stored as the fifth element of `squares`.
- After the loop is done, the vector `squares` is no longer empty and instead consists of the squares of the integers from 1 through 5: 1, 4, 9, 16, and 25.

```
#create empty vector to store results (initialize)
squares = vector("numeric", 5)

#run the loop
for(k in 1:5){

  squares[k] = k^2

}

#print the results
squares
```

```
## [1]  1  4  9 16 25
```

Of course, the same result could be easily achieved without a `for` loop:

```
(1:5)^2
```

```
## [1]  1  4  9 16 25
```

The `for` loop is a useful tool when the set of instructions to be repeated is more complicated, such as in the coin tossing scenario from the lab. The loop from Question 2 is reproduced here for reference.

- The loop is set to run for every k from 1 to the value of `number.replicates`, which has been previously defined as 50.
- There are two vectors defined within the curly braces.
  - The first vector, `outcomes.replicate`, consists of the outcomes for the tosses in a single replicate (i.e., iteration of the loop). The number of tosses in a single experiment has been defined as 5. These values are produced from `sample()`. This vector's values are re-populated each time the loop is run.
  - The second vector, `outcomes`, is created by calculating the sum of `outcomes.replicate` for each iteration of the loop. Once the loop has run, `outcomes` is a record of the number of `1`'s that occurred for each iteration.
- In the language of the coin tossing scenario: the loop starts by tossing 5 coins, counting how many heads occur, then recording that number as the first element of `outcomes`. Next, the loop tosses 5 coins again and records the number of heads in the second element of `outcomes`. The loop stops once it has repeated the experiment (of tossing 5 coins) 50 times.

```
for(k in 1:number.replicates){

  outcomes.replicate = sample(c(0, 1), size = number.tosses,
                              prob = c(1 - prob.heads, prob.heads), replace = TRUE)

  outcomes[k] = sum(outcomes.replicate)

}
```

`rep()`

The `rep(x, times)` function is a generic way to replicate elements of a vector `x` a certain number of times. In the lab, it is used to build a vector containing the elements that can be sampled from for the scenario of drawing differently colored balls from a bag. The following code creates a vector `balls` that contains 5 `R`'s and 2 `W`'s.

```
balls = rep(c("R", "W"), c(5, 2))
balls
```

```
## [1] "R" "R" "R" "R" "R" "W" "W"
```

Explicitly listing the elements to sample from is typically only necessary when the scenario calls for sampling without replacement. If, for example, the balls were to be drawn with replacement, then the sampling could simply be done from a vector `(R, W)` where there is probability 5/7 of drawing an `R` and 2/7 of drawing a `W`.

`if` Statements

An `if` statement has the basic structure `if ( condition ) { statement }` ; if the condition is satisfied, then the statement will be carried out.

The following loop prints the message "x is greater than 5" if the condition `x > 5` is satisfied; if it is not satisfied, then no message will be printed.

```
x = 100/10

if(x > 5){

  print("x is greater than 5")

}
```

```
## [1] "x is greater than 5"
```

Counting successes

Questions 4 and 5 from the lab introduce the structure of `if` statements in the context of calculating joint probabilities, such as the joint probability of drawing a white ball on the first pick and a red on the second. The `if` statements are used to record instances of successes, i.e., instances when both events of interest occurred.

These `if` statements are nested within `for` loops. The loop from Question 4 is reproduced here for reference.

- In the `if` statement, the condition requires that the first element of the vector `draw` must be a `"W"` and the second element an `R`.
- If the condition is satisfied for the k^{th} replicate, then a `1` is recorded as the k^{th} element of the vector `successes`.
- The joint probability can then be calculated by dividing the number of successes by the total number of replicates.

```
for(k in 1:replicates){

  draw = sample(balls, size = number.draws, replace = FALSE)

  if(draw[1] == "W" & draw[2] == "R"){

    successes[k] = 1

  }
}
```

In some cases, the event of interest can be more directly expressed with programming language than with an algebraic approach.

For example, Question 5 asks about the probability of drawing exactly one red ball. The algebraic approach involves using the logic that there are two scenarios in which exactly one red ball can be observed (red on the second draw or the first draw) and adding together the associated two joint probabilities.

This outcome can be expressed the same way in the condition of the `if` statement:

```r
if( (draw[1] == "W" & draw[2] == "R") | (draw[1] == "R" & draw[2] == "W") ){

  successes[k] = 1

}
```

However, it can also be expressed more simply with the help of the `sum()` function:

```r
if(sum(draw == "R") == 1){

  successes[k] = 1

}
```

Simulating populations

Question 6 uses `if` statements to simulate populations based on known conditional probabilities. This approach is useful when probabilites other than those provided are of interest, such as joint probabilities or the reverse conditional probabilities.

The loop from Question 6 is reproduced here for reference.

- The problem statement provides the conditional probabilities of being tall given sex. Height status must be assigned based on sex; `if` statements are used to specify the conditioning.
- Prior to running the loop, sex was randomly assigned using `sample()` and stored in the vector `sex`, where `0` represents a male and `1` represents a female.
- The vector `tall` will contain the height status of the simulated individuals.
- The first `if` statement has the condition `sex[k] == 0`. If the k^{th} element of `sex` is 0, then the following instructions are used to determine the k^{th} element of `tall`: sampling from `(0, 1)` with the probabilities `1 - p.tall.if.male` and `p.tall.if.male`, respectively.
- The second `if` statement has the condition `sex[k] == 1`. If the k^{th} element of `sex` is 1, then the sampling is done with the probabilities `1 - p.tall.if.female` and `p.tall.if.female`.
- Height status is assigned one at a time as the loop moves from `k in 1:population.size`, so `size = 1` for the `sample()` commands within the `if` statements. When `sample()` is used to assign sex, all the sampling is done in a single step, so `size = population.size`.

```r
for(k in 1:population.size){

  if(sex[k] == 0){

    tall[k] = sample(c(0,1), prob = c(1 - p.tall.if.male, p.tall.if.male),
                     size = 1, replace = TRUE)
  }

  if(sex[k] == 1){

    tall[k] = sample(c(0,1), prob = c(1 - p.tall.if.female, p.tall.if.female),
                      size = 1, replace = TRUE)
  }
}
```

The simulated population can be used to estimate joint probabilities such as the probability of being tall and female, or a conditional probability not given in the problem statement like $P(F|T)$.

```
#probability of female and tall
sum(sex == 1 & tall == 1)/population.size
```

```
## [1] 0.017
```

```
#probability of female given tall
sum(sex == 1 & tall == 1)/sum(tall == 1)
```

```
## [1] 0.143339
```

Distribution Functions

Binomial Distribution Functions

The function **`dbinom()`** used to calculate $P(X = k)$ has the generic structure

```
dbinom(x, size, prob)
```

where `x` is k, `size` is the number of trials n, and `prob` is the probability of success p.

The following code shows how to calculate $P(X = 5)$ for $X \sim \text{Bin}(10, 0.35)$. It is not necessary to explicitly specify the names of the arguments.

```
#probability X equals 5
dbinom(x = 5, size = 10, prob = 0.35)     #argument names specified
```

```
## [1] 0.1535704
```

```
dbinom(5, 10, 0.35)                       #argument names omitted
```

```
## [1] 0.1535704
```

The function **`pbinom()`** used to calculate $P(X \leq k)$ or $P(X > k)$ has the generic structure

```
pbinom(q, size, prob, lower.tail = TRUE)
```

where `q` is k, `size` is the number of trials n, and `prob` is the probability of success p. By default, R calculates $P(X \leq k)$. In order to compute $P(X > k)$, specify `lower.tail = FALSE`.

The following code shows how to calculate $P(X \leq 5)$ and $P(X > 5)$ for $X \sim \text{Bin}(10, 0.35)$.

```
#probability X is less than or equal to 5
pbinom(5, 10, 0.35)
```

```
## [1] 0.9050659
```

```
#probability X is greater than 5
pbinom(5, 10, 0.35, lower.tail = FALSE)
```

```
## [1] 0.09493408
```

The function **`rbinom()`** simulates a set number of draws from a binomial distribution and has the generic structure

```
rbinom(n, size, prob)
```

where `n` is the number of draws, `size` is the number of trials n, and `prob` is the probability of success p. Be sure to use `set.seed()` if it is of interest for the random draws to be reproducible.

Normal Distribution Functions

The function **`pnorm()`** used to calculate $P(X \leq k)$ or $P(X > k)$ has the generic structure

```r
pnorm(q, mean = 0, sd = 1, lower.tail = FALSE)
```

where `q` is k, `mean` is the parameter μ, and `sd` is the parameter σ. By default, R calculates $P(X \leq k)$ and assumes that mean and standard deviation are 0 and 1, respectively. In order to compute $P(X > k)$, specify `lower.tail = FALSE`.

The following code show show to calculate $P(X \leq 105)$ and $P(X > 105)$ for $X \sim \mathrm{N}(100, 5)$ and $P(Z \leq 1)$ and $P(Z > 1)$ for $Z \sim \mathrm{N}(0, 1)$.

```r
#probability X is less than (or equal to) 105
pnorm(105, 100, 5)
```

```
## [1] 0.8413447
```

```r
#probability X is greater than 105
pnorm(105, 100, 5, lower.tail = FALSE)
```

```
## [1] 0.1586553
```

```r
#probability Z is less than (or equal to) 1
pnorm(1)
```

```
## [1] 0.8413447
```

```r
#probaiblity Z is greater than 1
pnorm(1, lower.tail = FALSE)
```

```
## [1] 0.1586553
```

The function **`qnorm()`** used to identify the observation that corresponds to a particular probability p has the generic structure

```r
qnorm(p, mean = 0, sd = 1, lower.tail = FALSE)
```

where `p` is p, `mean` is the parameter μ, and `sd` is the parameter σ. By default, R identifies the observation that corresponds to area p in the lower tail (i.e., to the left) and assumes that mean and standard deviation are 0 and 1. To identify the observation with area p in the upper tail, specify `lower.tail = FALSE`.

The following code shows how to calculate the value of the observation (unstandardized and standardized) where there is 0.841 area to the left (and 0.159 area to the right).

```r
#identify X value
qnorm(0.841, 100, 5)
```

```
## [1] 104.9929
```

```r
qnorm(0.159, 100, 5, lower.tail = FALSE)
```

```
## [1] 104.9929
```

```r
#identify Z value
qnorm(0.841)
```

```
## [1] 0.9985763
```

```r
qnorm(0.159, lower.tail = FALSE)
```

```
## [1] 0.9985763
```

The function **`rnorm()`** simulates a set number of draws from a normal distribution and has the generic structure

```r
rbinom(n, mean = 0, sd = 1)
```

where `n` is the number of draws, `mean` is the parameter μ, and `sd` is the parameter σ. Be sure to use `set.seed()` if it is of interest for the random draws to be reproducible.