Tutorial 1: Probability Theory and Distributions

Jan Vogler (jan.vogler@duke.edu)

August 28, 2015

Today's Agenda

1. Useful R commands
2. For loops
3. Basic functions
4. Combinations and permutations
5. Basic probability
6. Conditional probability

1. Useful R commands

The following commands are relevant for the problem set and future tutorials.

Commands to create sequences and vectors:

```
1:10 # Creates a sequence of integers
```

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

```
seq(1,10) # Does the same thing
```

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

```
seq(2,50,by=2) # But you can specify the distance between elements
```

```
##  [1]  2  4  6  8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46
## [24] 48 50
```

```
newvector=c(1,2,3,4,5) # Creates a vector with five elements, same as 1:5
newvector
```

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

```
rep(1,500) # Creates a vector with the number 1 repeated 500 times
```

```
##   [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [36] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [71] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [106] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [141] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [176] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [211] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [246] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [281] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [316] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [351] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [386] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [421] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [456] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [491] 1 1 1 1 1 1 1 1 1 1
```

```r
fun=c("R","is","fun") # Creates a vector with non-numerical entries
paste(fun, collapse=" ") # Turns the above vector into a single element
```

```
## [1] "R is fun"
```

Important basic built-in functions:

```r
abs(-10)  # Returns the absolute value of a number
```

```
## [1] 10
```

```r
sqrt(4)  # Gives you the square root of a number
```

```
## [1] 2
```

```r
log(exp(1))  # Gives you the natural logarithm, here ln(e) was entered e = exp(1)
```

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

```r
log(100, base = 10)  # Gives you the logarithm of 100 with base 10
```

```
## [1] 2
```

```r
round(1.5)  # Rounds the value to the next integer
```

```
## [1] 2
```

```r
sum(newvector)  # Gives you the sum of all elements of a vector
```

```
## [1] 15
```

```r
length(newvector)  # Gives you the length (number of elements) of a vector
```

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

```
mean(newvector)  # Gives you the mean or average of a vector
```

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

```
sum(newvector)/length(newvector)  # Does the same thing (formula behind the above command)
```

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

```
summary(newvector)  # Gives you more detailed summary statistics
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##       1       2       3       3       4       5
```

```
which(newvector >= 3)  # Tells you which entries of the vector meet the condition of being larger than
```

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

If you have a logical vector, it can have two types of entries, TRUE (T) and FALSE (F).

TRUE and FALSE are possible values when conditions are formulated, such as "if" conditions within functions or loops.

```
logical = c(T, F, T, F, F)  # TRUE=T, FALSE=F
!logical  # Will reverse the vector
```

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

If you have a vector, you can exclude elements from it.

```
a = c(1, 2, 3, 4, 5)
b = a[-3]  # This command excludes the third element from the vector
b  # Only 1, 2, 4, and 5 are left
```

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

You can also exclude multiple elements from a vector with the following command.

```
c = a[-c(4, 5)]  # This command excludes the fourth and fifth element
c  # Only 1, 2, and 3 are left
```

```
## [1] 1 2 3
```

Important: this command does not exclude values, it excludes items by their position!

```
d = c(11, 15, 9, 2, 34)
e = d[-c(1, 2)]
e  # Only the last three items are left: 9, 2, and 34
```

```
## [1]  9  2 34
```

You can access individual elements of a vector with the following commands:

```
a[3]  # Gives you the third element of the vector
```

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

```
logical[2]  # Gives you the second element of the vector
```

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

You can also combine two vectors to a matrix.

```
matrix = rbind(a, logical)  # Rbind stands for 'rowbind', you bind two rows together, the alternative i
matrix  # The logical vector is turned numeric (1 for TRUE, 0 for FALSE)
```

```
##         [,1] [,2] [,3] [,4] [,5]
## a          1    2    3    4    5
## logical    1    0    1    0    0
```

```
matrix[2, 3]  # Gives you the element in the second row and third column
```

```
## logical
##       1
```

2. For loops

A for loop applies a specific action to a number of elements, indexed by a letter or combination of letters. The letter "i" is most frequently used for constructing for loops.

Example: We can use a for loop to calculate the sum of the numbers from 1 to 100.

```
number = 0  # First define number to be '0'
for (i in 1:100) {
    number = i + number
}
number  # After the for loop has been applied, it returns the value of the element number, which is 505
```

```
## [1] 5050
```

What if we only want to add the even numbers between 1 and 100?

We create an if condition inside the for loop.

```
number = 0
for (i in 1:100) {
    if (i%%2 == 0) {
        number = i + number
    }
}
number  # Returns the value of number = 2550
```

```
## [1] 2550
```

3. Basic functions

Unlike a for loop where you specify the elements for which the action is executed, a function is a general formula that then can be applied to different numbers.

This means the input of a function is not defined in advance. When you use the function later, you plug some values in.

```
addition = function(a, b) {
    print(a + b)
}

addition(1, 3)  # Returns the value 4
```

```
## [1] 4
```

Just like for loops, functions can also include conditions.

Let's create the same function but it's limited to the numbers between 0 and 10 for "a".

```
addition2 = function(a, b) {
    if (0 <= a & a <= 10) {
        print(a + b)
    } else {
        print("Error: The first input ('a') must be a number between 0 and 10.")
    }
}

addition2(5, 11)  # Returns 16
```

```
## [1] 16
```

```
addition2(11, 5)  # Returns our error message
```

```
## [1] "Error: The first input ('a') must be a number between 0 and 10."
```

Let's look at a slightly more complicated function.

Let us assume you want to plug in a positive number n > 3. You want to see how many numbers up to this number can be divided by 3 without a remainder. How would you do that?

```
divideby3 = function(number) {
    successes = rep(0, number)  # We first create a zero vector with the same length as the number that
    for (i in 1:number) {
        if (i%%3 == 0) {
            successes[i] = 1  # Every time a number meets the condition, we turn this entry in the zero
        }
    }
    sum(successes)  # We finally look at the sum all elements of the vector, where every '1' stood for
}

divideby3(6)
```

```
## [1] 2
```

```
divideby3(10)
```

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

This function is not entirely what we want because it allows for numbers <= 3 to be plugged in.

In order to account for this, we have to adjust the function and include an if condition.

```
divideby3 = function(number) {
    if (number > 3) {
        successes = rep(0, number)
        for (i in 1:number) {
            if (i%%3 == 0) {
                successes[i] = 1
            }
        }
        sum(successes)
    } else {
        print("The number has to be greater than 3.")
    }
}

divideby3(3)  # Now the function tells us that it has to be greater than 3
```

```
## [1] "The number has to be greater than 3."
```

4. Combinations and Permutations

Combinations and permutations fundamentally rest on the use of factorials.

In R, factorials are written in the following way:

```
factorial(3)  # 3! = 3*2*1 = 6
```

```
## [1] 6
```

```
factorial(5)  # 5! = 5*4*3*2*1 = 120
```

```
## [1] 120
```

There is a second factorial function in R, called the logarithmic factorial.

This function is used for high value factorials because they become hard to store.

```
lfactorial(2)  # log(2*1) = log(2) + log(1) = 0.6931472
```

```
## [1] 0.6931472
```

In order to write a permutation with the factorial command, we need to know “n” and “k”. Let n=10 and k=7.

```r
factorial(10)/factorial(3)  # This is the expression for 10!/3! = 604800
```

```
## [1] 604800
```

In order to write a combination with the factorial command, we need to account for the different orders in which the elements can occur. In the case of n=10 and k=7, there are 7! different combinations in which the elements can occur.

```r
factorial(10)/(factorial(3) * factorial(7))  # This is the expression for 10!/(3!*7!) = 120
```

```
## [1] 120
```

As you can see, there are much fewer combinations than permutations, why is this the case?

R has a built-in command for combinations.

```r
choose(10, 3)  # If we did everything right, it should return the same value '120'
```

```
## [1] 120
```

5. Basic probability

Let us create a vector that contains all possible outcomes of a die roll.

```r
die = c(1, 2, 3, 4, 5, 6)
```

What is the sample space in this case?

Now let's sample without replacement.

```r
sample(die, size=1) # Gives you a random number from the die
```

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

```r
sample(die, size=2) # Gives you two random numbers from the die
```

```
## [1] 5 4
```

```r
sample(die, size=7) # Returns an error, why?
```

```
## Error in sample.int(length(x), size, replace, prob): cannot take a sample larger than the population
```

Without running the code, what outcome would you expect from the following command?

```r
sample(die, size=6)
```

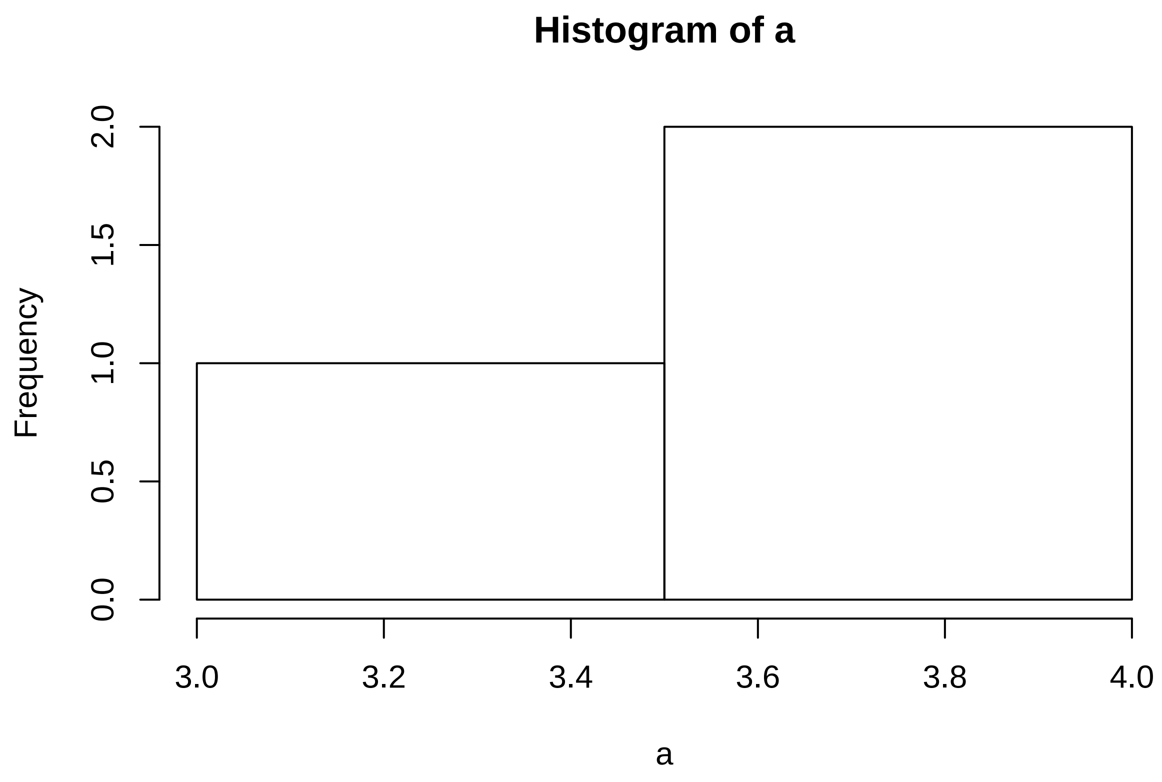
In order to make sure that you can draw n > 6, you need to replace a number after drawing it.

```r
sample(die, size = 10, replace = T)
```

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

Let's create several samples and look at their distribution.

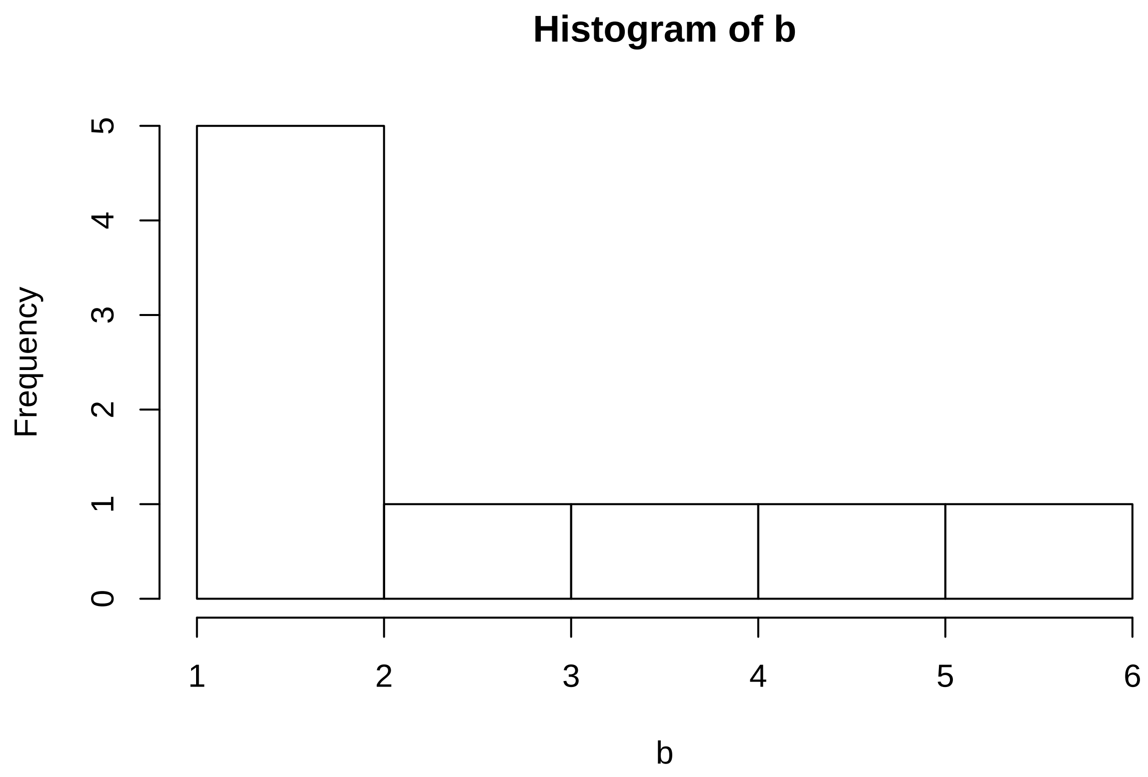
```r
a = sample(die, size = 3, replace = T)
hist(a)   # Gives you a histogram
```



```r
mean(a)
```

```
## [1] 3.666667
```

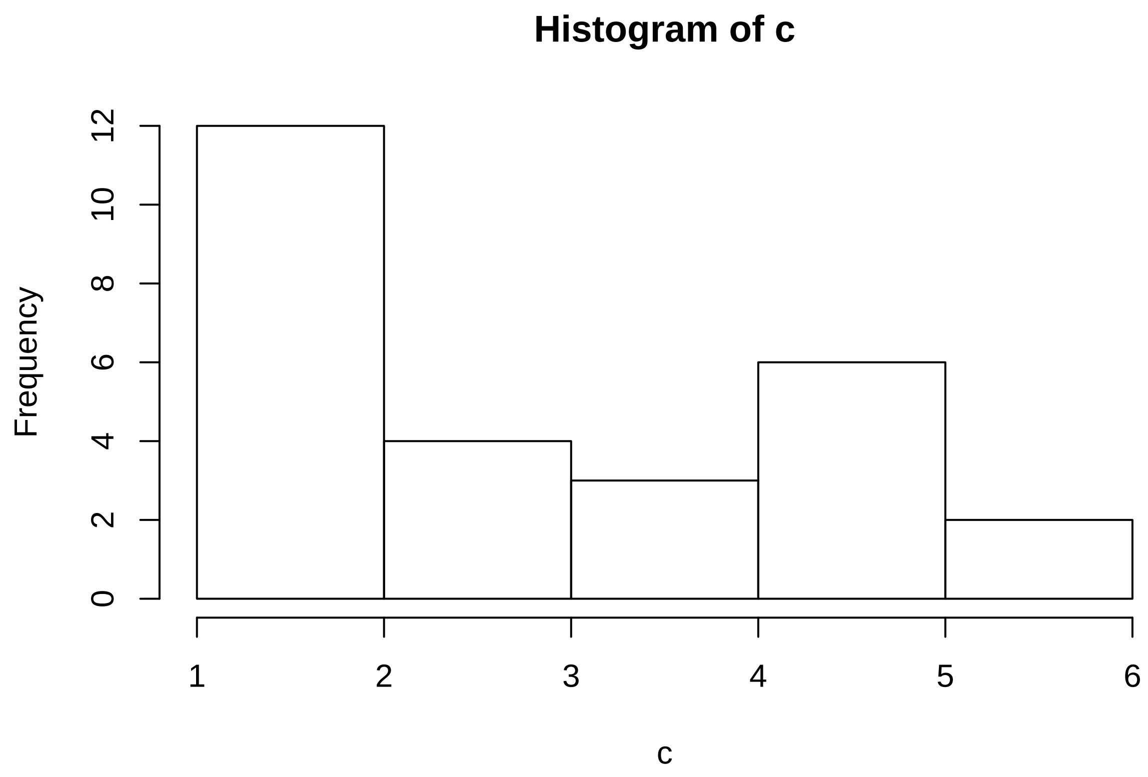
```r
b = sample(die, size = 9, replace = T)
hist(b)
```



```r
mean(b)
```

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

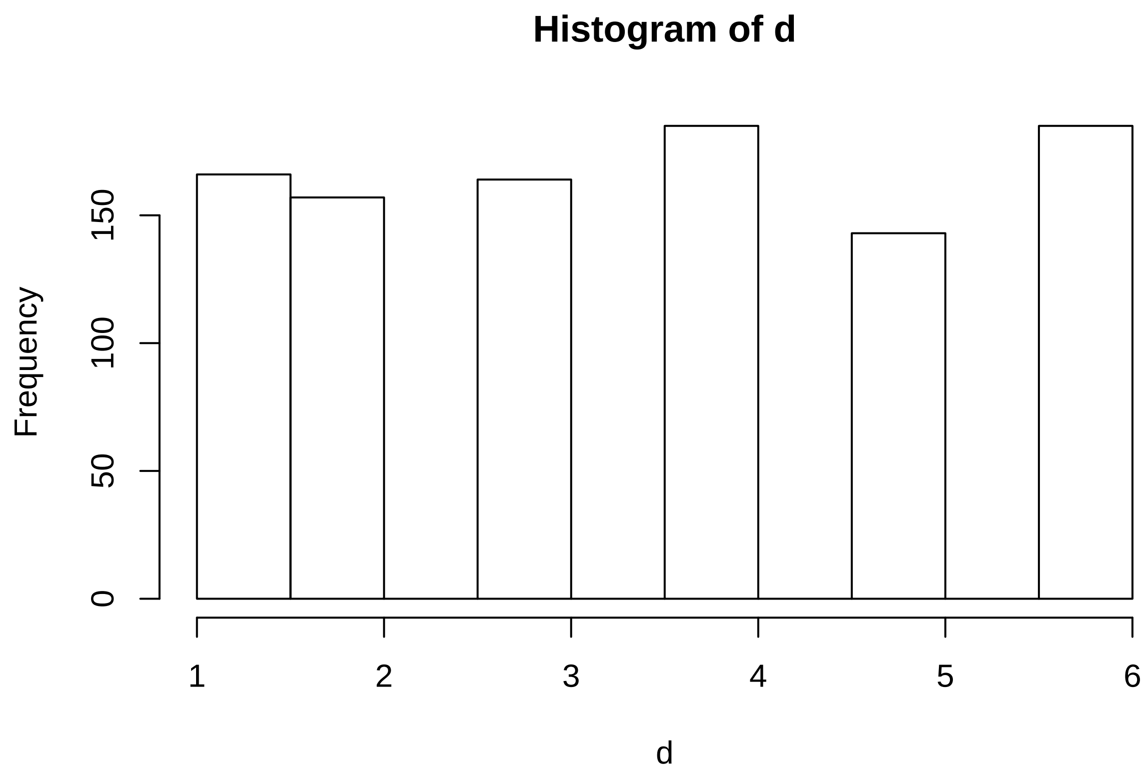
```r
c = sample(die, size = 27, replace = T)
hist(c)
```



```
mean(c)
```

```
## [1] 3.148148
```

```
d = sample(die, size = 1000, replace = T)
hist(d)
```



```r
mean(d)
```

```
## [1] 3.537
```

Looking at plot "d", what can you tell about the distribution of the die roll?

Let us define an event, A, which is that you draw an even number. What is the theoretical probability of A - Pr(A)?

How would we write a function that allows us to input a number of rolls and returns the proportion of rolls in which event A happens?

```r
x = which(d%%2 == 0)  # This command allows you to find all elements that have a remainder of 0 when di
length(x)/1000
```

```
## [1] 0.527
```

```r
evennum = function(rolls) {
    die = c(1, 2, 3, 4, 5, 6)
    sample1 = sample(die, size = rolls, replace = T)
    event = which(sample1%%2 == 0)
    proportion = length(event)/rolls
    print(proportion)
}

evennum(1000)  # Our theoretical expectation would be 0.5
```

```
## [1] 0.506
```

Let us define a second event B, which is that you draw a number smaller than 4

How would we write a function for that?

```
smaller4 = function(rolls) {
    die = c(1, 2, 3, 4, 5, 6)
    sample1 = sample(die, size = rolls, replace = T)
    event = which(sample1 < 4)
    proportion = length(event)/rolls
    print(proportion)
}

smaller4(1000)  # Our theoretical expectation would be 0.5
```

```
## [1] 0.483
```

Remember that Pr(A union B) = Pr(A) + Pr(B) - Pr(A intersection B). Let us try to find Pr(A union B) for events A and B as defined above.

Theoretically all numbers but 5 meet our condition, so theoretically we would expect the Pr(A union B) to be 5/6 ~ 0.833.

Let us write an (overly) complex function that illustrates all of this.

```
probunionAB = function(rolls) {
    die = c(1, 2, 3, 4, 5, 6)
    sample1 = sample(die, size = rolls, replace = T)
    event1 = which(sample1%%2 == 0)  # This is A
    event2 = which(sample1 < 4)  # This is B
    event3 = which(sample1%%2 == 0 & sample1 < 4)  # This is A intersection B
    proportion = (length(event1)/rolls + length(event2)/rolls - length(event3)/rolls)
    print(proportion)
}

probunionAB(1000)  # Returns a number very close to 0.833 for large n
```

```
## [1] 0.83
```

6. Conditional Probability

Next we will consider the Monty Hall problem. This will be part of the homework.

There are three possible locations of the prize:

```
prizeoptions = c(1, 2, 3)  # Define a vector that has all three locations in them - door 1, 2, and 3
prize = sample(prizeoptions, size = 1)  # Randomly draw a location of the prize
prize  # Where is it?
```

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

Let us assume that you also choose a door at random at first:

```r
doorchosen = sample(prizeoptions, size = 1)  # You can draw from the same vector because you also draw
doorchosen
```

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

What happens if your door equals the location of the prize? Monty Hall will show you one of the other two doors at random

If doorchosen=prize, the following happens:

```r
doorshown = sample(prizeoptions[-prize], size = 1)
doorshown
```

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

What happens if your door does not have the prize in it? That means your door is an empty door. Then, Monty Hall will show you the only remaining door that is also empty. Think about how that could be written in R code, it will be part of the homwork.

For your homework: Write a function in which you input the number of trials. The function should execute the above steps and always make you switch to the other door after Monty has shown you an emtpy one.

The function should give you the proportion of times that you were right in "n" trials. This won't be easy, you have to apply most things you have learned today.