Module 2

This is a single, concatenated file, suitable for printing or saving as a PDF for offline viewing. Please note that some animations or images may not work.

Module 2 Study Guide and Deliverables

Topics:	Probability
Readings:	• Lecture material
Assignments:	• Assignment 2 due **Tuesday, May 23 at 6:00 AM ET**
Assessments:	• Quiz 2 due **Tuesday, May 23 at 6:00 AM ET**
Live Classrooms:	• **Wednesday, May 17 from 7:00-9:00 PM ET** • One-hour open office with facilitator. Day and time will be provided by your facilitator. • Live sessions will be recorded.

Probability

Probability

The probability is a numeric measure that represents the chance or likelihood that a particular event will occur. The value ranges from \(0\) (impossible event) to \(1\) (certain event). The probabilities can be a priori, empirical, or subjective.

If the probabilities are based on prior knowledge of the process involved, these are referred to as *a priori* probabilities. Some examples are tossing a fair coin, drawing from a standard deck of cards, rolling a fair die, etc. When the outcomes are equally likely, the probability is defined as follows:

$$\text{Probability of event}=\frac{\text{Number of ways event can occur }(f)}{\text{Total number of possible outcomes }(N)}$$

Loading [Contrib]/a11y/accessibility-menu.js

If the probabilities are based on observed data, these are referred to as *empirical* probabilities. If \(70\%\) of the students own a PC, \(20\%\) own a Mac, and \(10\%\) do not have a computer, the probability that a student owns a PC is \(0.7\). Subjective probabilities are based on past experience and personal intuition and differ from person to person.

The collection of all possible outcomes for an experiment is known as the *sample space*. In the equal-likelihood model, all outcomes have the same chance of occurring and the probability of each outcome is the reciprocal of the size of the sample space.

An *event* is a collection of outcomes for an experiment.

Consider the experiment of rolling a pair of balanced dice. The sample space consists of \(36\) equally likely outcomes as shown below.

Die 2		**\(1\)**	**\(2\)**	**\(3\)**	**\(4\)**	**\(5\)**	**\(6\)**
Die 1	**\(1\)**	\((1,1)\)	\((1,2)\)	\((1,3)\)	\((1,4)\)	\((1,5)\)	\((1,6)\)
	\(2\)	\((2,1)\)	\((2,2)\)	\((2,3)\)	\((2,4)\)	\((2,5)\)	\((2,6)\)
	\(3\)	\((3,1)\)	\((3,2)\)	\((3,3)\)	\((3,4)\)	\((3,5)\)	\((3,6)\)
	\(4\)	\((4,1)\)	\((4,2)\)	\((4,3)\)	\((4,4)\)	\((4,5)\)	\((4,6)\)
	\(5\)	\((5,1)\)	\((5,2)\)	\((5,3)\)	\((5,4)\)	\((5,5)\)	\((5,6)\)
	\(6\)	\((6,1)\)	\((6,2)\)	\((6,3)\)	\((6,4)\)	\((6,5)\)	\((6,6)\)

Since all the outcomes are equally likely, the probability for any one of these outcomes is \(1/36\). The probabilities for some of the events are shown below.

Event	**Outcomes**	**Probability of the event**
Sum of dice is 1	None	0
Sum of dice is 12 or less	All	1
Sum of dice is 6	(1,5),(2,4),(3,3),(4,2),(5,1)	5/36
Two dice are equal	(1,1),(2,2),(3,3),(4,4),(5,5),(6,6)	6/36 = 1/6

Second die is greater than the first die	(1,2),(1,3),(1,4),(1,5),(1,6), (2,3),(2,4),(2,5),(2,6), (3,4),(3,5),(3,6), (4,5),(4,6), (5,6)	15/36

\(P(E)\) represents the probability of an event \(E\). If \(A\) and \(B\) are mutually exclusive events (i.e., they have no outcomes in common), then the probability of the event \(A\) or \(B\) is sum of the individual probabilities:

\(P(A\text{ or }B)=P(A)+P(B)\), where the events \(A\) and \(B\) are mutually exclusive.

The probability \(P(A \text{ or }B)\) is also written as \(P(A\cup B)\).

If the events \(A\) and \(B\) are not mutually exclusive, adding the probabilities of the two events will add the common outcomes \((A\&B)\) twice.

\(P(A\text{ or }B) = P(A)+P(B)–P(A\& B)\), where \(A\) and \(B\) are not mutually exclusive.

The probability \(P(A\text{ and }B)\) is also written as \(P(A\cap B)\).

For any event \(E\), the complement of the event is represented by *not \(E\)*.

\(P(\text{not }E)=1–P(E)\)

In the pair of dice example, let \(A\) be the event that the sum of dice is \(6\), and let \(B\) be the event that the two dice are equal. The event \((A \& B)\) has only one outcome \(\{(3,3)\}\). For this example,

\(P(A \& B) = 1/36\).

Hence, \(P(A\text{ or }B)=P(A)+P(B)–P(A\& B)=5/36+6/36–1/36=10/36\).

Similarly, the probability that the sum of the dice is not equal to \(6\) is

\(P(\text{not }A)=1–P(A)=1–5/36=31/36\).

The *conditional probability*, \(P(B|A)\), is the probability that event \(B\) occurs given that event \(A\) occurs. It is read as the probability of \(B\) given \(A\). Event \(A\) is called the given event.

Consider the roll of a single die with the six possible outcomes \(\{1,2,3,4,5,6\}\). Let \(A\) be the event that a \(4\) is rolled, and let \(B\) be the event that the die comes up even.

\(P(A) = 1/6 = 0.167, P(B) = 3/6 = 0.5\).

Given the die comes up even, the possible outcomes are only \(\{2,4,6\}\). The conditional probability, \(P(A|B)\), is the probability that a \(4\) is rolled given the die comes up even.

\(P(A|B) = 1/3 = 0.33\)

Given the die comes up even, there is a \(33\%\) chance of getting a \(4\) compared to a \(16.7\%\) chance of getting a \(4\) unconditionally.

\(P(B|\text{not }A)\) is the conditional probability that the die comes up even given that a \(4\) is not rolled. If a \(4\) is not rolled, there are only \(5\) possible outcomes \(\{1,2,3,5,6\}\). Hence, the conditional probability that the die co mes up even, given a \(4\) is not rolled, is

\(P(B|\text{not }A) = 2/5 = 0.4\).

There is a \(40\%\) chance in this case compared to \(50\%\) chance of getting an even value unconditionally.

If \(A\) and \(B\) are any two events, and \(P(A)\gt0\), then the *conditional probability* rule applies.

$$P(B|A)=\frac{P(A\&B)}{P(A)}$$

In the above example, the event \(A \& B\) has one outcome \(\{4\}\), hence \(P(A\&B) = 1/6\).

$$P(A|B)=\frac{P(A\&B)}{P(B)}=\frac{1/6}{3/6}=\frac{1}{3}$$

$$P(B|A)=\frac{P(A\&B)}{P(A)}=\frac{1/6}{1/6}=1$$

From the conditional probability rule,

$$P(A\&B)=P(A)\cdot P(B|A)$$

Event \(B\) is independent of the event \(A\) if \(P(B|A) = P(B)\). For independent events \(A\) and \(B\),

$$P(A\&B)=P(A)\cdot P(B)$$

Test Yourself 2.1

Compute the probability of rolling two dice and having the sum of the dice be greater than 6.

The outcomes are {(1,6), (2,5),(2,6), (3,4),(3,5),(3,6), (4,3),(4,4),(4,5),(4,6), (5,2),(5,3),(5,4),(5,5),(5,6), (6,1),(6,2),(6,3),(6,4),(6,5),(6,6)}
So, there are 21 possible outcomes out of 36. Hence, the probability is 21/36 = 0.58

Test Yourself 2.2

Compute the probability of rolling two dice and getting exactly one 5 in the roll.

The outcomes are {(5,1),(5,2),(5,3),(5,4),(5,6), (1,5),(2,5),(3,5),(4,5),(6,5)} So, there are 10 possible outcomes out of 36. Hence, the probability is 10/36 = 0.28

Test Yourself 2.3

Compute the probability that from a standard deck of cards, you will draw a J, Q, K, or a *Heart* on the first draw. Assume no Jokers in the deck.

Loading [Contrib]/a11y/accessibility-menu.js

The total number of outcomes are 3 * 4 + 10 = 22 (J, Q, K from the 4 suits Clubs, Diamonds, Hearts, and Spades + {A, 2, 3, …, 10} from the Hearts). Hence, the probability is 22/52 = 0.42

Installation of Required Packages

```
install.packages("combinat")
install.packages("https://cran.r-project.org/src/contrib/Archive/prob/prob_0.9-2.tar.gz",
                 repos = NULL, dependencies = TRUE)

library(prob)
Prob <- prob::prob
```

Sample Space

The set of all possible outcomes of a random experiment is called the *sample space*. We will denote the sample space using the letter \(S\). For example, in the case of tossing a coin once, the sample space \(S=\{H,T\}\), \(H\) denoting the outcome being the *head* and \(T\) the outcome being the *tail*. Similarly, if the coin is tossed twice, the sample space \(S=\{HH,HT,TH,TT\}\).

The `prob` package in *R* provides some common sample spaces like tossing coins, rolling dice, deck of cards, etc. First, let us examine some of the predefined sample spaces.

```
>library(prob)
```

The `tosscoin` function sets up the sample space for the experiment of tossing the coin repeatedly. The function returns a data frame object. The columns for the data frame are named `toss1`, `toss2`, …. The sample spaces for a single toss, two tosses, and three tosses are shown below.

```
>tosscoin(1)      >tosscoin(3)
 toss1             toss1 toss2 toss3
1    H            1   H     H     H
2    T            2   T     H     H
>tosscoin(2)      3   H     T     H
  toss1 toss2     4   T     T     H
1     H     H     5   H     H     T
2     T     H     6   T     H     T
3     H     T     7   H     T     T
4     T     T     8   T     T     T
```

The `rolldie` function sets up the sample space for the experiment of rolling a die repeatedly. The default for the number of sides of the die is \(6\). The function returns a data frame object. The columns for the data frame are named `X1`, `X2`

Loading [Contrib]/a11y/accessibility-menu.js

The following shows the sample spaces for a single roll of a \(6\)-sided die and a \(4\)-sided die respectively. The first argument is the number of rolls.

```
>rolldie(1) >rolldie(1, nsides = 4)
  X1            X1
1  1          1  1
2  2          2  2
3  3          3  3
4  4          4  4
5  5
6  6
```

The problems in this module require the use of the `prob` package. Use `library(prob)` to load the package.

Test Yourself 2.4

Show the *R* commands for the space of outcomes of tossing 4 coins, and rolling 2 six-sided dice

tosscoin(4)

rolldie(2)

Sampling from an Urn

Sampling from an urn of distinguishable objects is one of the common random experiments in statistics. If the sampling involves more than a single object, the following variations are possible:

- Ordered sampling with replacement
- Ordered sampling without replacement
- Unordered sampling with replacement
- Unordered sampling without replacement

In the ordered sampling, the order in which the objects are selected is important. In the unordered sampling, the order of the selections does not matter. For sampling with replacement, the selected object is put back and sampled again. For sampling without replacement, the selected object is not put back.

The `urnsamples` function provides the sampling from urns. The arguments for the function are:

- `x`—a vector or data frame from which sampling is done
- `size`—the sampling size
- `replace`—whether sampling should be done with replacement (default is `FALSE`)
- Loading [Contrib]/a11y/accessibility-menu.js er among samples is important (default is `FALSE`)

If the urn argument (x) is a vector, the function returns a data frame. The columns are named `X1`, `X2`,... If the urn is a data frame, the function returns a list of selections.

Suppose the urn has three balls labeled \(1\), \(2\), and \(3\) or colored red, green and blue. The following examples show the possible outcomes for a sample size \(2\) when sampling is done by default, i.e., unordered and done without replacement.

```
>urncamples(1:3, size = 2)
  X1  X2
1  1   2
2  1   3
3  2   3

>urnsamples(c("r", "g", "b"), size = 2)
  X1  X2
1  r   g
2  r   b
3  g   b
```

With replacement, the possible unordered outcomes are shown below.

```
>urnsamples(1:3, size = 2,
+     replace = TRUE)
  X1  X2
1  1   1
2  1   2
3  1   3
4  2   2
5  2   3
6  3   3

>urnsamples(c("r", "g", "b"), size = 2,
+   replace = TRUE)
  X1  X2
1  r   r
2  r   g
3  r   b
4  g   g
5  g   b
6  b   b
```

With ordering and replacement, the outcomes are shown below. For example, \((2, 1)\) and \((1, 2)\) are separate outcomes.

```
>urnsamples(1:3, size = 2,
+   replace = TRUE, ordered = TRUE)
  X1  X2
1  1   1
2  2   1
3  3   1
4  1   2
5  2   2
6  3   2
7  1   3
8  2   3
9  3   3
```

Loading [Contrib]/a11y/accessibility-menu.js

```
', "b"), size = 2,
+   replace = TRUE, ordered = TRUE)
```

```
   X1  X2
1   r   r
2   g   r
3   b   r
4   r   g
5   g   g
6   b   g
7   r   b
8   g   b
9   b   b
```

The last scenario shows ordered sampling without replacement.

```
>urnsamples(1:3, size = 2,
+   replace = FALSE, ordered = TRUE)
  X1  X2
1  1   2
2  2   1
3  1   3
4  3   1
5  2   3
6  3   2

>urnsamples(c("r", "g", "b"), size = 2,
+   replace = FALSE, ordered = TRUE)
```

Test Yourself 2.5

Many probability problems can be simulated by drawing proverbial objects from an urn.

Use the following vector to answer the following questions. Show all R commands

`seq(1:10)`

Use `urnsamples()` to draw 3 elements from the vector, with replacement, where the order does not matter.

Draw 2 elements from the vector, without replacement, where the order does matter

Draw 5 elements, without replacement, where the order does not matter

Draw 4 elements, with replacement, where the order does matter.

Which of these samples has the greatest number of possible outcomes?

x <- seq(1:10)

urnsamples(x, size=3, replace=TRUE, ordered=FALSE)

urnsamples(x, size=2, replace=FALSE, ordered=TRUE)

urnsamples(x, size=5, replace=FALSE, ordered=FALSE)

urnsamples(x, size=4, replace=TRUE, ordered=TRUE)

drawing 4, ordered, with replacement, produces the largest outcome space

Counting Tools

Loading [Contrib]/a11y/accessibility-menu.js

When the sample spaces are very large, counting methods can be used to compute the number of outcomes. Given \(n\), the number of distinguishable objects, and \(k\), the sample size, the following four scenarios are applicable for counting the number of outcomes:

- Ordered samples of size \(k\), with replacement—\(n\cdot n \cdot\ldots\cdot n(k \text{ times})=n^k\)
- Ordered samples of size \(k\), without replacement—\(n\cdot(n-1)\cdot\ldots\cdot(n-k+1)=\frac{n!}{(n-k)!}\)
- Unordered samples of size \(k\), without replacement—same as combinations of \(n\) distinct objects, taken \(k\) at a time = \(\frac{n!}{k!(n-k)!}=\binom{n}{k}=\binom{n}{n-k}\)
- Unordered samples of size \(k\), with replacement—\(\binom{n+k-1}{k}=\binom{n+k-1}{n-1}=\frac{(n+k-1)!}{k!(n-1)!}\)

The `nsamp` function calculates the above values, which is the number of rows in a sample space created by `urnsamples`, without explicitly generating the probability space. For an urn with \(3\) distinguishable elements, the number of ways for selecting \(2\) elements with the possible options for replacement and ordering are shown below. The default values are `FALSE` for `replace` and `ordered` arguments.

```
> nsamp(n = 3, k = 2, replace = FALSE, ordered = FALSE)
[1] 3
>
> nsamp(n = 3, k = 2, replace = TRUE, ordered = FALSE)
[1] 6
>
> nsamp(n = 3, k = 2, replace = FALSE, ordered = TRUE)
[1] 6
>
> nsamp(n = 3, k = 2, replace = TRUE, ordered = FALSE)
[1] 9
```

Test Yourself 2.6

From the previous problem, we saw instances where the sample space, or outcome space, was very large. Sometimes, we only care about the size of the space, rather than each individual element. Use the `nsamp()` function to illustrate the size of the sample spaces seen in the previous problem. Show all *R* commands.

nsamp(n=10, k=3, replace=TRUE, ordered=FALSE)

220

nsamp(n=10, k=2, replace=FALSE, ordered=TRUE)

90

nsamp(n=10, k=5, replace=FALSE, ordered=FALSE)

252

nsamp(n=10, k=4, replace=TRUE, ordered=TRUE)

10000

Loading [Contrib]/a11y/accessibility-menu.js

An event represents a subset of the sample space. An event is a collection of outcomes. In the generated sample space, an event is a selection of the appropriate rows.

The following sample space shows all the outcomes when a coin is tossed three times. The `makespace` option with the value `TRUE` creates the probability space for the outcomes by adding the `probs` column to the data frame. The sample space has a total of \(8\) outcomes. The probability for each outcome is \(1/8 = 0.125\).

```
> S <- tosscoin(3, makespace= TRUE)
> S
  toss1 toss2 toss3 probs
1     H     H     H 0.125
2     T     H     H 0.125
3     H     T     H 0.125
4     T     T     H 0.125
5     H     H     T 0.125
6     T     H     T 0.125
7     H     T     T 0.125
8     T     T     T 0.125
```

The `[]` operator can be used for filtering the rows of the sample space. Different approaches for selecting the rows are shown below.

```
> S[2:4, ]
  toss1 toss2 toss3 probs
2     T     H     H 0.125
3     H     T     H 0.125
4     T     T     H 0.125

> S[seq(1,8, by = 2), ]
  toss1 toss2 toss3 probs
1     H     H     H 0.125
3     H     T     H 0.125
5     H     H     T 0.125
7     H     T     T 0.125

> S[c(2,4,6,8), ]
  toss1 toss2 toss3 probs
2     T     H     H 0.125
4     T     T     H 0.125
6     T     H     T 0.125
8     T     T     T 0.125
```

The `subset` function can be used for selecting rows based on a logical condition. The following selects all rows where the third toss is a *head*.

```
> subset(S, toss3 == 'H')
  toss1 toss2 toss3 probs
1     H     H     H 0.125
2     T     H     H 0.125
3     H     T     H 0.125
4     T     T     H 0.125
```

The probability that the third toss is a *head* is \(4/8 = 0.5\). The sum of the `probs` column will also result in the same value.

Similarly, the outcomes where the first toss and the third toss are *heads* are selected as shown below.

Loading [Contrib]/a11y/accessibility-menu.js

```
> subset(S, toss1 == 'H' & toss3 == 'H')
  toss1 toss2 toss3 probs
1     H     H     H 0.125
3     H     T     H 0.125
```

The probability that the first toss and the third toss are *heads* is \(2/8 = 0.25\).

The following sample space shows all outcomes when a die is rolled twice. The `makespace` option with the value `TRUE` creates the probability space for the outcomes by adding the `probs` column to the data frame. The number of rows of the data frame is the total number of outcomes.

```
> S <- rolldie(2, makespace = TRUE)
>
> nrow(S)
1 [36]
```

The event that the two rolls are the same is the collection of outcomes filtered as shown below.

```
> subset(S, X1 == X2)
   X1 X2      probs
1   1  1 0.02777778
8   2  2 0.02777778
15  3  3 0.02777778
22  4  4 0.02777778
29  5  5 0.02777778
36  6  6 0.02777778
```

The probability that the two rolls are the same is \(6/36 = 0.167\).

The event that the sum of the two rolls is at least \(10\) is the collection of outcomes filtered as shown below.

```
> subset(S, X1 + X2 >= 10)
   X1 X2      probs
24  6  4 0.02777778
29  5  5 0.02777778
30  6  5 0.02777778
34  4  6 0.02777778
35  5  6 0.02777778
36  6  6 0.02777778
```

The probability that the sum of the two rolls is at least \(10\) is \(6/36 = 0.167\).

The event that the first roll is either \(5\) or \(6\) is the collection of outcomes filtered as shown below. The `%in%` function tests whether the values of the first argument lie anywhere within the second argument (`X1 %in% 5:6`). This is more convenient than specifying `X1 == 5 | X1 == 6`.

```
> subset(S, X1 %n% 5:6)
   X1 X2      probs
5   5  1 0.02777778
6   6  1 0.02777778
11  5  2 0.02777778
12  6  2 0.02777778
17  5  3 0.02777778
18  6  3 0.02777778
23  5  4 0.02777778
24  6  4 0.02777778
29  5  5 0.02777778
```

Loading [Contrib]/a11y/accessibility-menu.js

```
35   5   6 0.02777778
36   6   6 0.02777778
```

The probability that the first roll is either \(5\) or \(6\) is \(12/36 = 0.33\).

The following filters only the outcomes where the first roll is either \(5\) or \(6\) and the second roll is either \(1\) or \(3\).

```
> subset(S, X1 %n% 5:6 & X2 %n% c(1,3))
   X1 X2      probs
5   5  1 0.02777778
6   6  1 0.02777778
17  5  3 0.02777778
18  6  3 0.02777778
```

The probability of getting \(5\) or \(6\) in the first roll and \(1\) or \(3\) in the second roll is \(4/36 = 0.11\).

The following sample space sets up all \(216\) outcomes when a die is rolled three times.

```
> S <- rolldie(3, makespace = TRUE)
>> nrow(S)
[1] 216
```

The `isin(x, y)` function is useful for checking whether all the values of \(y\) are in \(x\). By default, the *ordered* argument is `FALSE`. There is only one outcome where the sequences of rolls have to be \(4\), \(5\), and \(6\).

```
> subset(S, isin(S, c(4,5,6), ordered = TRUE))
    X1 X2 X3      probs
208  4  5  6 0.00462963
```

If ordering is not required, all permutations of \((4, 5, 6)\) are possible outcomes.

```
> subset(S, isin(S, c(4,5,6))
    X1 X2 X3      probs
138  6  5  4 0.00462963
143  5  6  4 0.00462963
168  6  4  5 0.00462963
178  4  6  5 0.00462963
203  5  4  6 0.00462963
208  4  5  6 0.00462963
```

The probability of getting \(4\), \(5\), and \(6\) in any order is \(6/216 = 0.0278\).

The second argument can be a subset. The following filter selects two values (\(4\) and \(6\)) that should occur in order among the three rolls.

```
> subset(S, isin(S, c(4,6), ordered = TRUE)
    X1 X2 X3      probs
34   4  6  1 0.00462963
70   4  6  2 0.00462963
106  4  6  3 0.00462963
142  4  6  4 0.00462963
178  4  6  5 0.00462963
184  4  1  6 0.00462963
190  4  2  6 0.00462963
196  4  3  6 0.00462963
199  1  4  6 0.00462963
200  2  4  6 0.00462963
                      53
202  4  4  6 0.00462963
```

Loading [Contrib]/a11y/accessibility-menu.js

```
203   5   4   6 0.00462963
204   6   4   6 0.00462963
208   4   5   6 0.00462963
214   4   6   6 0.00462963
```

The `cards()` function sets the sample space for the standard deck of cards. The default option for `jokers` argument is `FALSE`. The function returns a data frame with \(52\) rows. The two columns of the data frame are named `rank` and `suit`.

```
> S <- cards(makespace = TRUE)
>
> nrow(S)
[1] 52
>
> head(S, n = 2)
  rank suit      probs
1    2 Club 0.01923077
2    3 Club 0.01923077
```

The following subsets show the \(13\) cards for each suit.

```
> subset(S, suit == "Club")
  rank  suit      probs
1    2  Club 0.01923077
2    3  Club 0.01923077
3    4  Club 0.01923077
4    5  Club 0.01923077
5    6  Club 0.01923077
6    7  Club 0.01923077
7    8  Club 0.01923077
8    9  Club 0.01923077
9    10 Club 0.01923077
10   J  Club 0.01923077
11   Q  Club 0.01923077
12   K  Club 0.01923077
13   A  Club 0.01923077

> subset(S, suit == "Diamond")
  rank     suit      probs
14   2  Diamond 0.01923077
15   3  Diamond 0.01923077
16   4  Diamond 0.01923077
17   5  Diamond 0.01923077
18   6  Diamond 0.01923077
19   7  Diamond 0.01923077
20   8  Diamond 0.01923077
21   9  Diamond 0.01923077
22   10 Diamond 0.01923077
23   J  Diamond 0.01923077
24   Q  Diamond 0.01923077
25   K  Diamond 0.01923077
26   A  Diamond 0.01923077

> subset(S, suit == "Heart")
  rank   suit      probs
27   2  Heart 0.01923077
28   3  Heart 0.01923077
29   4  Heart 0.01923077
30   5  Heart 0.01923077
                     077
                     077
```

```
33   8  Heart 0.01923077
34   9  Heart 0.01923077
35   10 Heart 0.01923077
36   J  Heart 0.01923077
37   Q  Heart 0.01923077
38   K  Heart 0.01923077
39   A  Heart 0.01923077

> subset(S, suit == "Spade")
  rank  suit     probs
40   2  Spade 0.01923077
41   3  Spade 0.01923077
42   4  Spade 0.01923077
43   5  Spade 0.01923077
44   6  Spade 0.01923077
45   7  Spade 0.01923077
46   8  Spade 0.01923077
47   9  Spade 0.01923077
48   10 Spade 0.01923077
49   J  Spade 0.01923077
50   Q  Spade 0.01923077
51   K  Spade 0.01923077
52   A  Spade 0.01923077
```

The outcomes where the rank of the selected card is \(2\), \(3\), or \(4\) can be filtered as shown below.

```
> subset(S, rank %n% 2:4)
   rank     suit     probs
1     2     Club 0.01923077
2     3     Club 0.01923077
3     4     Club 0.01923077
14    2  Diamond 0.01923077
15    3  Diamond 0.01923077
16    4  Diamond 0.01923077
27    2    Heart 0.01923077
28    3    Heart 0.01923077
29    4    Heart 0.01923077
40    2    Spade 0.01923077
41    3    Spade 0.01923077
42    4    Spade 0.01923077
```

The probability that the selected card is \(2\), \(3\), or \(4\) is \(12/52 = 0.23\).

The outcomes where the selected card is a *King* or a *Queen* can be filtered as shown below.

```
> subset(S, rank %n% c('K', 'Q'))
   rank     suit     probs
11    Q     Club 0.01923077
12    K     Club 0.01923077
24    Q  Diamond 0.01923077
25    K  Diamond 0.01923077
37    Q    Heart 0.01923077
38    K    Heart 0.01923077
50    Q    Spade 0.01923077
51    K    Spade 0.01923077
```

The probability that the selected card is a *King* or a *Queen* is \(8/52 = 0.15\).

The set operations `union`, `intersect`, and `setdiff` perform the union, the intersection, and the difference of the

Loading [Contrib]/a11y/accessibility-menu.js

For the card deck sample space \(S\), let the event \(A\) represent the selected card suit is a *Heart* and the event \(B\) represent the selected card rank is either a *10* or a *Queen*.

```
> S <- cards(makespace = TRUE)
> A <- subset(S, suit == "Heart")
> B <- subset(S, rank %in% c(10, "Q"))
```

The possible outcomes for the events \(A\) and \(B\) are shown below.

```
> A
  rank  suit      probs
27   2 Heart 0.01923077
28   3 Heart 0.01923077
29   4 Heart 0.01923077
30   5 Heart 0.01923077
31   6 Heart 0.01923077
32   7 Heart 0.01923077
33   8 Heart 0.01923077
34   9 Heart 0.01923077
35  10 Heart 0.01923077
36   J Heart 0.01923077
37   Q Heart 0.01923077
38   K Heart 0.01923077
39   A Heart 0.01923077

> B
   rank    suit      probs
9    10    Club 0.01923077
11    Q    Club 0.01923077
22   10 Diamond 0.01923077
24    Q Diamond 0.01923077
35   10   Heart 0.01923077
37    Q   Heart 0.01923077
48   10   Spade 0.01923077
50    Q   Spade 0.01923077
```

The union of the events \(A\) and \(B\) consists of all the outcomes that are in \(A\) or \(B\) or both.

```
> union(A, B)
   rank    suit      probs
9    10    Club 0.01923077
11    Q    Club 0.01923077
22   10 Diamond 0.01923077
24    Q Diamond 0.01923077
27    2   Heart 0.01923077
28    3   Heart 0.01923077
29    4   Heart 0.01923077
30    5   Heart 0.01923077
31    6   Heart 0.01923077
32    7   Heart 0.01923077
33    8   Heart 0.01923077
34    9   Heart 0.01923077
35   10   Heart 0.01923077
36    J   Heart 0.01923077
37    Q   Heart 0.01923077
38    K   Heart 0.01923077
39    A   Heart 0.01923077
48   10   Spade 0.01923077
50    Q   Spade 0.01923077
```

Loading [Contrib]/a11y/accessibility-menu.js (A\) and \(B\) consists of all the outcomes that are in both \(A\) and \(B\).

```
> intersect(A, B)
   rank  suit      probs
35   10 Heart 0.01923077
37    Q Heart 0.01923077
```

The difference of the events \(A\) and \(B\) consists of all the outcomes that are in \(A\) but not in \(B\).

```
> setdiff(A, B)
27    2  Heart 0.01923077
28    3  Heart 0.01923077
29    4  Heart 0.01923077
30    5  Heart 0.01923077
31    6  Heart 0.01923077
32    7  Heart 0.01923077
33    8  Heart 0.01923077
34    9  Heart 0.01923077
36    J  Heart 0.01923077
38    K  Heart 0.01923077
39    A  Heart 0.01923077
```

The set difference operation is not commutative. The following shows the different results for the difference of the events \(B\) and \(A\).

```
> setdiff(B, A)
   rank    suit      probs
9    10    Club 0.01923077
11    Q    Club 0.01923077
22   10 Diamond 0.01923077
24    Q Diamond 0.01923077
48   10   Spade 0.01923077
50    Q   Spade 0.01923077
```

The complement of an event \(A\), written as \(A^{\mathsf{c}}\), is the difference between the sample space, \(S\), and the event \(A\). `setdiff(S, A)` will result in all outcomes that are either Clubs, Diamonds, or Spades.

Test Yourself 2.7

Use `rolldie()` to create the sample space of rolling 2 dice. Find the probability that an event will result in both dice having the same value. Find the probability that an event will result in the sum of both dice being even. Find the probability that the first die will be a 4 or a 5. Find the intersection of the second and third set of events described above. Find the union of the first and second set of events described above. Show the *R* commands.

▶ Show Hint

s <- rolldie(2, makespace=TRUE)

event1 <- subset(s, X1 == X2)

sum(event1$probs)

0.166667

event2 <- subset(s, (X1 + X2) % 2 == 0)

sum(event2$probs)

0.5

event3 <- subset(s, X1 %in% c(4, 5))

Loading [Contrib]/a11y/accessibility-menu.js)

0.33333

intersect(event2, event3)

union(event1, event2)

Setting up the Probability Space

All the sample spaces considered so far follow the equally likely model, where each outcome in the sample space has the same probability (reciprocal of the size of the sample space). For each outcome, a different probability may be assigned using the `probspace` function.

The following code shows an alternate approach for assigning probabilities for the experiment of rolling a die. The outcomes are generated as shown below.

```
> outcomes <- rolldie(1)
> outcomes
  X1
1  1
2  2
3  3
4  4
5  5
6  6
```

Since there are \(6\) outcomes, \(6\) probability values are needed. The following shows a vector of equal values.

```
> p <-rep(1/6, times = 6)
> p
[1] 0.1666667 0.1666667 0.1666667 0.1666667 0.1666667 0.1666667
```

The `probspace` function assigns the probabilities to each of the corresponding outcomes.

```
> probspace(outcomes, probs = p)
  X1     probs
1  1 0.1666667
2  2 0.1666667
3  3 0.1666667
4  4 0.1666667
5  5 0.1666667
6  6 0.1666667
```

The above is equivalent to the `rolldie` invocation with `makespace` argument as `TRUE`.

```
> rolldie(1, makespace= TRUE)
 X1       probs
1  1 0.1666667
2  2 0.1666667
```

Loading [Contrib]/a11y/accessibility-menu.js

```
5  5 0.1666667
6  6 0.1666667
```

Suppose that the die is not fair, with some faces having a better chance than the rest. The following example shows a vector of different probabilities used to set up the sample probability space.

```
> p <- c(0.2, 0.15, 0.15, 0.15, 0.15, 0.2)
> probspace(outcomes, probs = p)
  X1 probs
1  1  0.20
2  2  0.15
3  3  0.15
4  4  0.15
5  5  0.15
6  6  0.15
```

Test Yourself 2.8

Use `probspace()` to set up a probability space for rolling a loaded die, where the probability of rolling of a 3 or a 4 are 0.2, and the remaining numbers have an equal probability. Show the *R* commands

outcomes <- seq(1:6)

(1 - 2*(0.2)) / 4

p <- c(0.15, 0.15, 0.2, 0.2, 0.15, 0.15)

space <- probspace(space, p)

The *Prob* Function

The `Prob` function calculates the probability of the specified event. Consider the sample space from the deck of cards.

```
> S <- cards(makespace = TRUE)
```

The probability of drawing a *Queen* can be determined as follows: (Let \(A\) be the event for the outcomes representing drawing a *Queen.)*

```
> A <- subset(S, rank == "Q")
> A
   rank    suit      probs
11    Q    Club 0.01923077
24    Q Diamond 0.01923077
37    Q   Heart 0.01923077
50    Q   Spade 0.01923077
```

Loading [Contrib]/a11y/accessibility-menu.js ds, the probability is \(4/52 = 0.0769\).

```
> Prob(A)
[1] 0.07692308
>
> Prob(S, rank == "Q")
[1] 0.07692308
```

Let \(B\) be the event that the card is a *Heart*.

```
> B <- subset(S, suit == "Heart")
> B
  rank   suit      probs
27    2  Heart 0.01923077
28    3  Heart 0.01923077
29    4  Heart 0.01923077
30    5  Heart 0.01923077
31    6  Heart 0.01923077
32    7  Heart 0.01923077
33    8  Heart 0.01923077
34    9  Heart 0.01923077
35   10  Heart 0.01923077
36    J  Heart 0.01923077
37    Q  Heart 0.01923077
38    K  Heart 0.01923077
39    A  Heart 0.01923077
```

Since there are \(13\) cards, the probability of drawing a *Heart* is \(13/52 = 0.25\).

```
> Prob(B)
[1] 0.25
>
> Prob(S, Suit == "Heart")
[1] 0.25
```

The probability of the intersection of the events \(A\) and \(B\) (*Queen of Hearts*) is shown below.

```
> Prob(intersect(A, B))
[1] 0.01923077
```

The probability of the union of the events \(A\) and \(B\) (a *Queen* or a *Heart* or both) is shown below.

```
> Prob(union(A, B))
[1] 0.3076923
```

This is equivalent to the following.

```
> Prob(A) + Prob(B) - Prob(intersect(A, B))
[1] 0.3076923
```

Test Yourself 2.9

Use the `Prob()` function and `cards()` function to answer the following questions. Assume no jokers in the deck.

Loading [Contrib]/a11y/accessibility-menu.js ility of drawing a Queen or King from a deck of cards?

What is the probability of drawing a number between 3 and 5 inclusive that it is also a *Heart*?

What is the probability of the union of the two events above?

Show the R commands.

```
deck <- cards(makespace=TRUE)
event1 <- subset(deck, rank %in% c("Q", "K"))
prob(event1)
0.1538462
prob(deck, rank %in% c(3, 4, 5) & suit == 'Heart')
0.05769231
event2 <- subset(deck, rank %in% c(3, 4, 5) & suit == 'Heart')
prob(union(event1, event2))
0.2115385
```

Conditional Probability

Conditional Probability

The conditional probability rule, $P(B|A)$, computes the probability of event B given that event A has occurred.

$$P(B|A)=\frac{P(A\cap B)}{P(A)}$$

The following properties apply to conditional probabilities as well. For the sample space S and any fixed event A, with $P(A)\gt 0$,

- $P(B|A)\ge 0$, for all events $B\subset S$
- $P(S|A)=1$
- If $B_1,B_2,\ldots,B_k$ are disjoint events, then $P(B_1\cup B_2\cup\ldots\cup B_k|A)=P(B_1|A)+P(B_2|A)+\ldots + P(B_k|A)$
- $P(B^{\mathsf{c}}|A)=1–P(B|A)$, where B^{c} is the complement of the event B
- For any events A,B, and C, if $B\subset C$, then $P(B|A)\le P(C|A)$
- $P(B\cup C|A)= P(B|A)+P(C|A)–P(B\cap C|A)$, where B and C are not disjoint

The *multiplication rule* is derived from the conditional probability rule as follows:

$P(A\cap B)= P(A\text{ and } B)= P(A)\cdot P(B|A)$

Conditional Probability Example - Rolling Die Twice

Consider two rolls of a six-sided die. The sample space, $S=\{(1,1), (1,2), \ldots, (6,6)\}$. The size of the sample space is $\#(S)= 6*6= 36$.

The sample space generated in *R* is shown below. The first two and the last two outcomes along with the individual probabilities are also shown.

```
> S <- rolldie(2, makespace= TRUE)
> head(S, n = 2)
   X1 X2      probs
1   1  1 0.02777778
2   2  1 0.02777778
> tail(S, n = 2)
   X1 X2      probs
35  5  6 0.02777778
36  6  6 0.02777778
```

Let A denote the event where both the rolls have the same value. The possible outcomes for this event are:

$A=\{(1,1),(2,2),(3,3),(4,4),(5,5),(6,6)\}$

Hence, $P(A)= 6/36= 1/6= 0.167$

The event A is computed in *R* as shown below.

```
> A <- subset(S, X1 == X2)
> A
   X1 X2      probs
1   1  1 0.02777778
8   2  2 0.02777778
15  3  3 0.02777778
22  4  4 0.02777778
29  5  5 0.02777778
36  6  6 0.02777778
> Prob(A)
[1] 0.1666667
```

Let B be the event where the sum of the two outcomes is equal to 8. The possible outcomes for this event are:

$B= \{(6,2),(5,3),(4,4),(3,5),(2,6)\}$.

Hence, $P(B)= 5/36= 0.139$

The event B is computed in *R* as shown below.

```
> B <- subset(S, X1 + X2 == 8)
> B
   X1 X2      probs
12  6  2 0.02777778
```

Loading [Contrib]/a11y/accessibility-menu.js

```
27   3   5 0.02777778
32   2   6 0.02777778
> Prob(B)
[1] 0.1388889
```

The event \(A\cap B\) in this example has only one outcome \(\{(4,4)\}\).

Hence, \(P(A\cap B)= 1/36\).

The conditional probabilities in this example are

\(P(AIB)= P(A\cap B)/P(B)= (1/36)/(5/36)= 1/5= 0.2\), and

\(P(BIA)= P(A\cap B)/P(A)= (1/36)/(6/36)= 1/6= 0.167\)

Using *R*, the conditional probabilities are computed as shown below.

```
> Prob(A, given = B)
[1] 0.2
>
> Prob(B, given = A)
[1] 0.1666667
```

The same could be done in *R* in a few steps without explicitly defining the subsets \(A\) and \(B\), as follows:

```
> S <- rolldie(2, makespace= TRUE)
> Prob(S, X1 == X2, given = (X1 + X2 == 8))
[1] 0.2
> Prob (S, X1 + X2 == 8, given = (X1 == X2))
[1] 0.1666667
```

Conditional Probability Example - Coin Toss Twice

Consider two tosses of a coin. The sample space, \(S=\{(H,H),(H,T),(T,H),(T,T)\}\). The size of the sample space is \(\#(S)=4\).

The sample space generated in *R* is shown below. The four outcomes along with the individual probabilities are also shown.

```
> S <- tosscoin(2, makespace = TRUE)
> S
  toss1 toss2 probs
1     H     H  0.25
2     T     H  0.25
3     H     T  0.25
4     T     T  0.25
```

Let \(A\) be the event that a *head* occurs. The possible subset outcomes for this event are:

Loading [Contrib]/a11y/accessibility-menu.js

Hence, $P(A)=3/4=0.75$

The event A is computed in *R* as shown below.

```
> A <- subset(S, isin(S, c('H')))
> A
  toss1 toss2 probs
1     H     H  0.25
2     T     H  0.25
3     H     T  0.25
```

Let B be the event that a *head* and a *tail* occurs. The possible outcomes for this event are:

$B=\{(H,T),(T,H)\}$.

Hence, $P(B)=2/4=0.5$

The event B is computed in *R* as shown below.

```
> B <- subset(S, isin(S, c('H', 'T')))
> B
  toss1 toss2 probs
2     T     H  0.25
3     H     T  0.25
```

The event $A\cap B$ in this example has two outcomes $\{(H,T),(T,H)\}$.

Hence, $P(A\cap B)=2/4$.

The conditional probabilities in this example are:

$$\begin{align}P(A|B)&=P(A\cap B)/P(B)\\&=(2/4)/(2/4)\\&=1/1\\&=1\end{align}$$

From the above, once we know that a *head* and *tail* occur, it is certain that a *head* occurs. Similarly:

$$\begin{align}P(B|A)&=P(A\cap B)/P(A)\\&=(2/4)/(3/4)\\&=2/3\\&=0.667\end{align}$$

Using *R*, the conditional probabilities are computed as shown below.

```
> Prob(A , given = B)
[1] 1
>
> Prob(B, given = A)
[1] 0.6666667
>
```

Conditional Probability Example - Card Deck

Consider the standard full deck of 52 playing cards. If selecting two cards from that deck in sequence, let the

is an ace} and B = {second card drawn is an ace}.

For the first card, there are 4 aces, hence $P(A)=4/52$.

The probability for the second card being an ace depends on whether the first card is an ace or not. If the first card is an ace, then the probability of the second card also being an ace is $3/51$. However, if the first card is not an ace, then the probability of the second card being an ace is $4/51$.

$P(B|A)=3/51$, and $P(B|A^{\mathsf{c}})= 4/51$.

By the multiplication law, the probability of both cards being aces is

$P(A\cap B)= P(A)\cdot P(B|A)= (4/52)\cdot (3/51)= 0.00452$.

For generating the sample space in *R* for the above problem (drawing two cards), the *cards* function is first used to model the deck of cards as shown below. The data frame has two columns, *rank* and *suit*. The first four rows are also shown.

```
> L <- cards()
> head(L, n = 4)
  rank suit
1    2 Club
2    3 Club
3    4 Club
4    5 Club
```

The `urnsamples` function can be used to sample two cards randomly from the above data frame. The function with a size of 2 returns all possible pairs of rows. The first three pairs generated by this function are shown below. The sample space for our problem is this list M.

```
> M <- urnsamples(L, size = 2)
> head(M, n = 3)
[[1]]
  rank suit
1    2 Club
2    3 Club

[[2]]
  rank suit
1    2 Club
3    4 Club

[[3]]
  rank suit
1    2 Club
4    5 Club
```

The above set consists of $C(52,2)= 1326$ pairs of cards. This can be verified in *R* by computing the length of the generated pairs as shown below.

```
> length(M)
[1] 1326
>
> choose(52, 2)
```

The probability model for the sample space is computed as shown below.

For the event that both cards are aces, the *rank* of both the cards should be “A”.

```
> S <- probspace(M)
>
> Prob(S, all(rank == "A"))
[1] 0.004524887
```

Similarly, the probability of both cards being *Clubs* is calculated as follows:

```
> Prob(S, all(suit == "Club"))
[1] 0.05882353
```

Conditional Probability Example - Red and Blue Balls

Consider a box with three red balls and two blue balls inside it. For the given problem of selecting two successive balls, we are interested in finding the probability that both the balls selected will be red. Let A be the event that the first ball is red, and B be the event that the second ball is also red.

A = {first ball is red}, B = {second ball is red}

$P(A)=3/5$, and $P(B|A)=2/4$.

Probability that both the balls are red

$$\begin{align}P(A\cap B)&=P(A)\cdot P(B|A)\\&=(3/5)\cdot (2/4)\\&= 3/10\\&= 0.3\end{align}$$

Similarly, the probability that both the balls are blue = $(2/5)\cdot (1/4) = 0.1$.

The problem can be modeled in *R* as follows: (The data now consists of one dimension, the color of the balls. The data is generated using the `rep` function with the *red* repeated three times and the *blue* repeated twice.)

```
> L <-rep(c("red", "blue"), times = c(3,2))
> L
[1] "red" "red" "red" "blue" "blue"
```

The `urnsamples` function can be used to generate all outcomes for selecting two balls randomly from the above list. With a size of two, the function returns all possible pairs of rows. Also, the ordered argument is set to `TRUE`, as the ordering of the balls is considered. The sample space for our problem is this list M.

```
> M <- urnsamples(L, size = 2, ordered = TRUE)
> M
     X1   X2
1    red  red
2    red  red
3    red  red
5    red blue
```

```
6  blue   red
7   red blue
8  blue   red
9   red   red
10  red   red
11  red blue
12 blue   red
13  red blue
14 blue   red
15  red blue
16 blue   red
17  red blue
18 blue   red
19 blue blue
20 blue blue
```

All possible outcomes of selecting two ordered pairs out of five balls are shown above. The probability space for the data along with the first two rows is computed as shown below.

```
> S <- probspace(M)
> head(S, n = 2)
   X1  X2 probs
1 red red  0.05
2 red red  0.05
```

Now, for the event that both the balls are red, all the rows with \(X_1\) == “red” & \(X_2\) == “red” satisfy the criteria. An alternative is the `isrep` function in the `prob` package. `isrep(S, “red”, 2)` tests each row of the sample space to determine whether the value *red* appears two times. The probabilities of both balls being *red* and both balls being *blue* are computed as shown below.

```
> Prob(S, isrep(S, "red", 2))
[1] 0.3
>
> Prob(S, isrep(S, "blue", 2))
[1] 0.1
```

The probability of the first ball being *red* and the second ball being *blue* can be calculated as shown below.

```
> Prob(S, isin(S, c("red", "blue"), ordered = TRUE))
[1] 0.3
```

Test Yourself 2.10 (Multipart)

(Click each step to reveal its solution.)

▶ Consider rolling a six-sided die 3 times. Let Event A represent the case where the sum of the values of three rolls is less than 10. Let Event B represent the case where the second roll is a 3. Using the `rolldie()` and `prob()` functions and whatever other functions needed, compute the probability of A, B, and A and B (intersection). Show the R commands

```
S <- rolldie(3, makespace=TRUE)
A <- subset(S, X1 + X2 + X3 < 10)
prob(A)
0.375
B <- subset(S, X2 == 3)
prob(B)
```

Loading [Contrib]/a11y/accessibility-menu.js

```
C <- intersect(A, B)
```

```
prob(C)
0.0694444
```

▶ **Using the probabilities from above, compute the conditional probability of B, given A, and the conditional probability of A, given B. Show the R commands.**

```
prob(B, given=A)
0.1851852
prob(A, given=B)
0.416667
```

▶ **Consider drawing from a deck of cards (no Jokers). Use the `cards()` and `urnsamples()` function to simulate drawing 2 cards without replacement. Let A be the event that you have at least one *Ace*, let B the event you have two cards of the same rank. Compute the probability of A, B, and A and B (intersection). Then compute the conditional probability of A, given B and the conditional probability of B, given A. Show the *R* commands.**

```
cards <- cards()
hand <- urnsamples(cards, size=2, replace=FALSE, ordered=FALSE)
S <- probspace(hand)
A <- subset(S, 'A' %in% rank)
prob(A)
0.1493213
B <- subset(S, rank[1] == rank[2])
prob(B)
0.05882353
C <- intersect(A, B)
prob(C)
0.004524887
prob(A, given=B)
0.07692308
prob(B, given=A)
0.03030303
```

Independent Events

Two events \(A\) and \(B\) are said to be independent if

$$P(A\cap B)=P(A)\cdot P(B)$$

From the conditional probability rule

$$P(B|A)=\frac{P(A\cap B)}{P(A)}$$

If the events \(A\) and \(B\) are independent, the occurrence of the event \(A\) has no effect on the event \(B\). In that case, \(P(B|A)=P(B)\):

$$P(B)=\frac{P(A\cap B)}{P(A)}$$

and thus \(P(B)\cdot P(A)=P(A\cap B)\).

Loading [Contrib]/a11y/accessibility-menu.js

If the events \(A\) and \(B\) are independent, then:

- A and B^{c} are independent
- A^{c} and B are independent
- A^{c} and B^{c} are independent

Three events A, B, and C are mutually independent if and only if:

- $P(A\cap B)=P(A)\cdot P(B)$
- $P(B\cap C)=P(B)\cdot P(C)$
- $P(A\cap C)=P(A)\cdot P(C)$
- $P(A\cap B\cap C)=P(A)\cdot P(B)\cdot P(C)$

Independent Events Example - Coin Toss

Consider the sample space for a coin tossed five times. Let $A_1, A_2, \ldots, A_5$ be the events that the corresponding coin toss is a head. In this case, $P(A_1)=\frac{1}{2}$ and $P(A_1^{\mathsf{c}})=\frac{1}{2}$, etc.

Since the events are mutually independent, the probability of all tosses being *tails* is:

$$\begin{align}P(A_1^{\mathsf{c}}\cap A_2^{\mathsf{c}}\cap A_3^{\mathsf{c}}\cap A_4^{\mathsf{c}}\cap A_5^{\mathsf{c}})&= P(A_1^{\mathsf{c}})\cap P(A_2^{\mathsf{c}})\cap P(A_3^{\mathsf{c}})\cap P(A_4^{\mathsf{c}})\cap P(A_5^{\mathsf{c}})\\&=\left(\frac{1}{2}\right)^5\\&= 0.03125\end{align}$$.

The probability of having at least one head in the five coin tosses then is:

$$\begin{align}P(\text{at least one head})&=1–P(\text{all tails})\\&=1-\left(\frac{1}{2}\right)^5\\&=0.96875\end{align}$$

The above example is modeled in *R* as shown below.

```
> S <- tosscoin(5, makespace=TRUE)
>
> B <- subset(isrep(S, "T", 5))
>
> B
   toss1 toss2 toss3 toss4 toss5   probs
32     T     T     T     T     T 0.03125
```

The probability of all *tails* and the probability of at least one *head* are:

```
> Prob(B)
[1] 0.03125
>
> 1 - Prob(B)
[1] 0.96875
```

Independent Events, Repeated Experiments

The iidspace function is used for modeling an experiment that is repeated multiple times under identical conditions in an independent manner. The function takes the following arguments:

- A vector of possible outcomes for the experiment in a single trial.
- The number of times the experiment is repeated.
- The vector of probabilities of the outcomes in a single trial. If this argument is not specified, all outcomes will have equal probability.

The following example shows the sample space for an unbalanced coin that is tossed three times. The coin is biased, favoring *heads*. Since the outcomes are independent, the probability for all three *heads* is:

\(\begin{align}P(X1=′H′\cap X2=′H′\cap X3 =′H′) &= P(X1 =′H′)\cdot P(X2 =′H′)\cdot P(X3 = ′H′)\\&=0.6 \cdot 0.6\cdot 0.6\\&= 0.216\end{align}\)

```
> iidspace(c('H', 'T'), ntrials = 3, probs = c(0.6, 0.4))
  X1 X2 X3 probs
1  H  H  H 0.216
2  T  H  H 0.144
3  H  T  H 0.144
4  T  T  H 0.096
5  H  H  T 0.144
6  T  H  T 0.096
7  H  T  T 0.096
8  T  T  T 0.064
```

When the individual probabilities are not specified, all outcomes have equal probability as shown below.

```
> iidspace(c('H', 'T'), ntrials = 3)
  X1 X2 X3 probs
1  H  H  H 0.125
2  T  H  H 0.125
3  H  T  H 0.125
4  T  T  H 0.125
5  H  H  T 0.125
6  T  H  T 0.125
7  H  T  T 0.125
8  T  T  T 0.125
```

Test Yourself 2.11

Consider rolling two fair 6-sided dice. Intuitively, we understand that the two die rolls are independent events. Let A be the event that the first roll is a 2, let B be the event that the second roll is also a 2. Show the R commands to illustrate the independence of the die rolls, specifically, show that P(A and B) == P(A) * P(B)

```
S <- rolldie(2, makespace=TRUE)
A <- subset(S, X1 == 2)
B <- subset(S, X2 == 2)
C <- intersect(A, B)
prob(A)
```

Loading [Contrib]/a11y/accessibility-menu.js

```
prob(B)
0.16667
prob(A) * prob(B)
0.027778
prob(C)
0.027778
```

Test Yourself 2.12

Using the `iidspace()` function, simulate the rolling of 5 fair 6-sided dice. Show the *R* commands for the probability of all rolls being greater than 3, the first roll is a 2, and the sequence: 1, 2, 3, 4, 5

```
S <- iidspace(c(1, 2, 3, 4, 5, 6), ntrials=5)
prob(S, X1 > 3 & X2 > 3 & X3 > 3 & X4 > 3 & X5 > 3)
.03125
prob(S, X1 == 2)
0.16667
prob(S, X1 == 1 & X2 == 2 & X3 == 3 & X4 == 4 & X5 == 5)
0.0001286008
```

Bayes' Rule

Bayes' rule is used for revising probabilities with newly acquired information. The rule of *total probability* is used in defining Bayes' rule.

Suppose $A_1,A_2,\ldots ,A_k$ are mutually exclusive and exhaustive events, i.e., exactly one of the events must occur. Then, for any event B, the events $(A_1\cap B),(A_2\cap B), \ldots,(A_k\cap B)$ are mutually exclusive, and

$$P(B)= P(A_1\cap B)+P(A_2\cap B)+\ldots + P(A_k\cap B)$$

Using the *multiplication rule*:

$$P(B)= P(A_1)\cdot P(B|A_1)+P(A_2)\cdot P(B|A_2)+\ldots +P(A_k)\cdot P(B|A_k)$$

In other words:

$$P(B)=\sum_{i=1}^{k}P(A_i)\cdot P(B|A_i)$$

For Bayes' rule, we assume that the probabilities $P(A_1),P(A_2),\ldots ,P(A_k)$ are known from the given data. Also, we assume that the conditional probabilities $P(B|A_1),P(B|A_2),\ldots ,P(B|A_k)$ are also known. Bayes' rule is concerned with the computation of the probabilities $P(A_1|B),P(A_2|B),\ldots ,P(A_k|B)$.

For any i, the probabilities $P(A_i|B)$ are computed as follows:

$$P(A_i|B)= \frac{P(A_i\cap B)}{P(B)}= \frac{P(A_i)\cdot P(B|A_i)}{P(B)}$$

Hence,

$$P(A_i|B)= \frac{P(A_i)\cdot P(B|A_i)}{\sum_{j=1}^{k}P(A_j)\cdot P(B|A_j)}$$

In Bayes' rule, the probabilities $P(A_i)$ are known as *prior* probabilities and the probabilities $P(A_i|B)$ are known as *posterior* probabilities. The probabilities $P(B|A_i)$ are the *likelihood* probabilities.

Bayes' Rule Example

Suppose 7% of the population has lung disease. Among those having lung disease, 90% are smokers. Of those not having lung disease, 25% are smokers. Bayes' rule is used to determine the probability that a randomly selected smoker has lung disease.

Let A_1 be the event that the person selected has lung disease and A_2 be the event that the person selected has no lung disease. The two events A_1 and A_2 are mutually exclusive and exhaustive. From the given data, the *prior* probabilities are

$$P(A_1)=0.07\text{ and }P(A_2)=1-P(A_1)=0.93$$

Let B be the event that the person selected is a smoker. From the given data, the *likelihood* probabilities are:

$$P(B|A_1)=0.9\text{ and }P(B|A_2)=0.25.$$

The probability that a randomly selected smoker has lung disease is:

$$\begin{align}P(A_1|B)&=\frac{P(A_1)\cdot P(B|A_1)}{P(A_1)\cdot P(B|A_1)+P(A_2)\cdot P(B|A_2)}\\&= \frac{0.07\cdot 0.9}{0.07\cdot 0.9+0.93\cdot 0.25}\\&=0.213\end{align}$$

The probability that a randomly selected smoker does not have lung disease is:

$$\begin{align}P(A_2|B)&= \frac{P(A_2)\cdot P(B|A_2)}{P(A_1)\cdot P(B|A_1)+P(A_2)\cdot P(B|A_2)}\\&= \frac{0.93\cdot 0.25}{0.07\cdot 0.9+0.93\cdot 0.25}\\&= 0.787\end{align}$$

Note that the two *posterior* probabilities sum to 1.

The same can be computed in *R* as follows:

```
> bayes <- function (prior, liklihood) {
+  numerators <- prior * likelihood
+  return (numerators / sum(numerators))
+ }
>
> prior <- c(0.07, 0.93)
> like <- c(0.9, 0.25)
> bayes(prior, like)
[1] 0.213198 0.786802
```

From the calculations, 21.3% of smokers have lung disease.

Test Yourself 2.13

A survey was taken of individuals aged 22-30. 40% of the individuals surveyed received a college degree. Of the individuals who had a college degree, 90% of them had parents who also received a college degree. Of the individuals who did not receive a college degree, only 20% of them had parents who received a college degree. First, compute the *prior* probabilities and the *likelihood* probabilities. Using the `bayes()` function defined below, compute the probability that randomly selected parents with a college degree have a child without a college degree. Show the R commands

```
bayes <- function(prior, likelihood) {
+     numerators <- prior * likelihood
+     return (numerators / sum(numerators))
+ }
```

Let A be the event an individual has a college degree

P(A) = 0.4

P(~A) = 0.6

Let B be the event the individual has parents with a college degree

P(B I A) = 0.9

P(B I ~A) = 0.2

We want to compute P(~A I B) = P(~A) * P(B I ~A) / (P(~A) * P(B I ~A) + P(A) / P(A) * P(B I A)

Substitute known values: 0.6 * 0.2 / (0.6 * 0.2 + 0.4 * 0.9)

prior = c(0.4, 0.6)

likelihood = c(0.9, 0.2)

bayes(prior, likelihood)

0.75, 0.25

25% chance

Basic Concepts of *R*—Programming Constructs

Functions and Function Arguments

A function takes zero or more arguments and returns a value. A function with a single argument can be written as shown below. The function is invoked with the required input values. The function can have an explicit `return` statement. Otherwise, the last statement is evaluated and returned.

Loading [Contrib]/a11y/accessibility-menu.js

```
                      x) {
+ returm (x + 1)
+ }
>
> inc.1(10)
[1] 11
>
> inc.1(c(10,20,30))
[1] 11 21 31
```

A function with two arguments is shown below. When the function is invoked, the parameter names can be explicitly assigned the input values. If the input values are named, then the inputs can be provided in any order.

```
> inc.2 <- function (x, y) {
+ return (x - y)
+ }
>
> inc.2(10,20)
[1] -10
> inc.2(x = 10, y = 20)
[1] -10
> inc.2(y = 20, x = 10)
[1] -10
```

The following example shows the mapping of the input values when only one of the arguments is named.

```
> inc.2(10, y = 20)
[1] -10
> inc.2(y = 20, 10)
[1] -10
```

The above function definitions expect two arguments to be provided when the function is invoked. If one or more inputs are missing, the function invocation throws an error. The error messages show that the arguments have no defaults, and hence expect a value to be provided.

```
> inc.2(10)
Error in inc.2(10) : argument "y" is missing, with no default
>
> inc.2(y = 20)
Error in inc.2(y = 20) : argument "x" is missing, with no default
```

The function arguments can be provided default values as shown below. In the following function definition, the value for `x` is required. The value for `y` is optional, the default being \(100\).

```
> inc.3 <- function (x, y = 100) {
+ return (x  + y)
+ }
>
> inc.3(10)
[1] 110
> inc.3(10, 20)
[1] 30
```

Invoking the above function with zero arguments throws an error that `x` is required.

```
> inc.3()
Error in inc.3() : argument "x" is missing, with no default
```

The following function definition has default values for both the arguments. When the function is invoked with no arguments, the defaults are used for both the arguments. In the last case, when a new value for `y` is provided, it has been explicitly named.

Loading [Contrib]/a11y/accessibility-menu.js

```
> inc.4 <- function (x = 5, y = 7) {
+ return (x + 7)
+ }
>
> inc.4()
[1] 12
> inc.4(2)
[1] 9
> inc.4(2,3)
[1] 5
> inc.4(y =3)
[1] 8
```

The following example shows a function with one required parameter and two optional parameters.

```
> inc.5 <- function (x, y = 100, z = 10) {
+ return (x - y - z)
+ }
>
> inc.5(1000)
[1] 890
> inc.5(1000, 500)
[1] 490
> inc.5(1000, 500, 200)
[1] 300
```

The function is invoked with different combinations as shown below.

```
> inc.5(1000, z = 200)
[1] 700

> inc.5(1000, z = 200, y = 500)
[1] 300
```

Test Yourself 2.14

Create a function called `sum` that takes two arguments, and returns the sum of the two. Call the function with the values (4, 5) and (6, 10). Try some negative values as well.

sum <- function(x, y) {

return (x + y)

}

Test Yourself 2.15

Create a function called `contains_item` that takes two arguments, a vector of `ints`, and a single `int` value. The function should return `TRUE` if the vector contains the int and `FALSE` otherwise. Call the function and test it out.

contains_item <- function(vec, x) {

return (x %in% vec)

}

Loading [Contrib]/a11y/accessibility-menu.js

Scope of Variables

When objects are assigned values in *R*, they belong to the scope of the global environment. Each function call creates a new frame that contains the scope for the arguments and any assignments to (local) variables within the function.

The following example demonstrates the scope of a single variable \(x\). The variable \(x\) is defined in the global environment.

```
x <- 10

foo <- function() {
   cat("#2 In foo ", x, "\n");
   x <- 20
   cat("#3 In foo ", x, "\n");
   bar()
   cat("#6 In foo ", x, "\n");

}

bare <- function() {
   cat("#4 In bar ", x, "\n");
   x <-30
   cat("#5 In bar ", x, "\n");
}
```

The function *foo* provides a new value for \(x\). However, this \(x\) will be in the local scope of the function *foo*. In step #2, when the value of \(x\) is used, the \(x\) refers to the value in the global scope (the value 10). In step #3, the value of \(x\) refers to the local definition (the value 20). From that point, all references of \(x\) within *foo* refer to the local definition. When the function *bar* is invoked from the function *foo*, a new frame for the function is created with its own local scope. The reference to \(x\) in step #4 refers to the global value of \(x\) (10). After \(x\) is assigned a new value (30), the local scope contains this definition of \(x\) and all references of \(x\) from that point (step #5) refer to this new definition. After the function *bar* returns, the value of \(x\) in step #6 is the one contained in *foo* (20). Finally, when the function *foo* returns, the value of \(x\) is from the global scope (10).

The output of the variable \(x\) in the different scopes of the above functions is shown below.

```
cat("#1 Before foo ", x, "\n");
foo()
cat("#7 After foo ", x, "\n");

> cat("#1 Before foo ", x, "\n");
#1 Before foo 10
> foo()
#2 In foo 10
#3 In foo 20
#4 In bar 10
#5 In car 30
#6 In foo 20
> cat("#7 After foo ", x, "\n");
#7 After foo 10
```

The following example shows the function with two arguments. The local scope for the function includes the two

Loading [Contrib]/a11y/accessibility-menu.js

function parameters (\(a\) and \(b\)) and the three variables defined locally (\(x,y\), and \(z\)).

```
> rm(z)
>
> x <- 10
> y <- 20
>
> test.1 <- function(a, b) {
+ x <- a
+ y <- b
+ z <- a + b
+ return (z)
+ }
```

After the function is invoked, \(x\) and \(y\) refer to the global scope; the object \(z\) defined within the function is no longer accessible.

```
> test.1(1,2)
[1] 3
>
> x
[1] 10
> y
[1] 20
> z
Error: object 'z' not found
```

The following example shows the access of variables from the local scope (\(a\)) and the global scope (\(x\)).

```
> x <- 20
> test.2 <- function(a) {
+ z <- a + x
+ return (z)
+ }
>
> test.2(1)
[1] 21
```

Within a function, the `<<-` operator can be used to change the value in the global scope. If the variable doesn't exist in the global scope, the `<<-` operator creates the variable in the global scope.

In the following example, the function changes the value of \(x\) in the global scope and inserts the variable \(z\) in the global scope.

```
> x <- 100
> y <- 200
>
> test.3 <- function(a, b) {
+ x <<- a
+ y <- b
+ z <<- x + y
+ return (z)
+ }
```

After the function returns, the modified values of \(x\) and \(z\) are available.

```
> test.3(1,2)
[1] 3
>
> x
```

Loading [Contrib]/a11y/accessibility-menu.js

```
[1] 200
> z
[1] 3
```

Control Structures

Some of the control structures provided in *R* are `if-else`, `for`, `while`, and `repeat`.

The `if-else` statement can be used in the following forms:

> **if** (*cond*) expr
>
> **if** (*cond*) expr1 *else* expr2

The following example compares two objects and computes the maximum/minimum of the two. The objects \(x\) and \(y\) are assumed to be scalars (vectors of unit length).

```
if (x < y) {
    max <- y
    min <- x
} else {
    max <- x
    min <- y
}
```

The `if-else` always returns the last expression from the `if` part or the `else` part. The following example shows the assignment of \(y\) or \(x\) to `max` based on the outcome of the test.

```
max <- if (x < y) y else x
```

The `for` loop has the following syntax.

> **for** (*name* **in** *values*) *expr*

The values can be a vector, a list, a matrix, or a data frame. For a vector, the loop iterates over each of the elements. For a matrix, the loop iterates over each of the elements column-wise. For a list, the loop iterates over the components in the list. Similarly, for the data frame, the loop iterates over the columns.

The following example shows the loop using a vector of values.

```
> x <- c(10, 20, 30)
>
> for (i in x) {
+ cat("Square of ", i, " = ", i*i, "\n")
+ }
Square of 10 = 100
Square of 20 = 400
Square of 20 = 900
```

The variable `i` retains the last value as a side effect.

```
> i
```

Loading [Contrib]/a11y/accessibility-menu.js

The loop can be used with any generated sequence of values.

```
> for (i in seq(1,10, by = 2)) {
+ cat("Square of ", i, " = ", i*i, "\n")
+ }
Square of 1 = 1
Square of 3 = 9
Square of 5 = 25
Square of 7 = 49
Square of 9 = 81
```

An example of the `while` loop for adding the first *n* numbers is shown below. The loop continues as long as the test is true.

```
> n <- 10
> sum <- 0
> i <- 1
>
> while (i <= n) {
+ sum < sum + i
+ i < i+1
+ }
> cat("Sum of first ", n, " numbers = ", sum)
Sum of first 10 numbers = 55
```

The `break` statement may be used to abort the loop. The following example calculates how many numbers add up to the limit and aborts the loop thereafter.

```
> limit <- 55
> sum <- 0
> i <- 0
>
> while (TRUE) {
+ i <- i+1
+ sum <- sum + i
+ if (sum >= limit) break
+ }
> cat("Sum of first ", i, " numbers = ", sum)
Sum of first 10 numbers = 55
```

An example of the `repeat` statement is shown below. An explicit `break` is required to come out of the loop.

```
> i <= 1
> repeat {
+ cat("Square of ", i, " = ", i*i, "\n")
+ i <- i+2
+ if (i > 10) break
+ }
Square of 1 = 1
Square of 3 = 9
Square of 5 = 25
Square of 7 = 29
Square of 9 = 81
```

Test Yourself 2.16

Create a function called `evens_only` that takes one argument, a vector of `ints`. The function

Loading [Contrib]/a11y/accessibility-menu.js

ctor containing only the even numbers in the vector. Use a `for` loop and `if-else` statements in the body of the function. Can you implement the function without using those control structures? Test out your functions a few times to ensure they work.

```
evens_only <- function (x) {
    holder = c()
        for (i in seq(1:length(x))) {
                if (x[i] %% 2 == 0) {
                        holder = c(holder, x[i])
                }
        }
        return (holder)
}

evens_only <- function (x) {
        return (x[x %% 2 == 0])
}
```

Reading & Writing Data

The `scan` function is useful for reading data from the console (or from a file or a network location). For reading numeric data from the console, the function may be used without any arguments. The reading terminates when a new line is entered on the console. The data read is returned as a vector.

```
> x <- scan()
1: 10 20 30
4: 40 50 60
7:
Read 6 items
> x
[1] 10 20 30 40 50 60
```

The white space delimiter separates the individual values. If any other delimiter is used, the `sep` argument is provided with the required delimiter.

```
> x <- scan(sep = ",")
1: 10,20,30
4: 40,50,60
7:
Read 6 items
> x
[1] 10 20 30 40 50 60
```

If the input is character data, the `what` argument is needed to specify the type of the data being read (default being `double()`).

```
> x <- scan(what = character())
1: Alice Bob Charlie
4: Dave Ed
6:
Read 5 items
>
[1] "Alice" "Bob" "Charlie" "Dave" "Ed"
```

A logical vector can be created by reading the `true/false` values as shown below.

```
> x <- scan(what = logical())
1: true
2: FALSE
3: false
4: TRUE
5:
Read 4 items
> x
[1] TRUE FALSE FALSE TRUE
```

A list can also be created with the same number of values for each component. The name and the type for each component is specified using the `list` option for the `what` argument. The data for the number of components is entered line by line.

```
> x <- scan(what =
+     list(age = numeric(), name = character()))
1: 30 Alice
2: 35 Bob
3L 25 Charlie
4:
Read 3 records
>
> x
$age
[1] 30 35 25

$name
[1] "Alice" "Bob" "charlie"
```

If the data for all the components is not entered in a single line, *R* prompts for the rest of the values on the subsequent line showing the corresponding index.

```
> x <- scan(what =
+      list(age = numeric(), name = character()))
+
1: 30
1: Alice
2: 35 Bob
3: 25
3: Charlie
4:
Read 3 records
>
> x
$age
[1] 30 35 25

$name
[1] "Alice" "Bob" "Charlie"
```

The `file` argument may be used with the `scan` function for reading the data from a file. The default value for this argument is the empty string that is interpreted as reading from the console. For reading from a file, if the full path is not specified, the current working directory is used for resolving the specified file name. The `getwd` function shows the current working directory.

Loading [Contrib]/a11y/accessibility-menu.js

```
[1] "/Users/skalathur"
```

If the data files are to be read from a specific directory, the `setwd` function can be used to change the working directory to the desired directory.

```
> setwd("/Users/skalathur/MyCourses/CS544/Samples/data")
```

The `dir` function lists the contents of the current working directory.

```
> dir()
[1] "athletedata.csv" "athletedata.txt"
```

The file `athletedata.txt` contains the following information regarding five athletes. The first row contains the names of the columns of the data.

```
Name        Salary Endorsements Sport
Mayweather  105.0  0            Boxing
Ronaldo     52.0   28           Soccer
James       19.3   53           Basketball
Messi       41.8   23           Soccer
Bryant      30.5   31           Basketball
```

The `scan` function can either read the data as a vector or as a list. Reading the entire contents as character data produces the vector as shown below.

```
> x <- scan("athletedata.txt", what=character())
Read 24 times
>
> x
 [1] "Name"       "Salary"      "Endorsements"
 [4] "Sport"      "Mayweather"  "105.0"
 [7] "0"          "Boxing"      "Ronaldo"
[10] "52.0"       "28"          "Soccer"
[13] "James"      "19.3"        "53"
[16] "Basketball" "Messi"       "41.7"
[19] "23"         "Soccer"      "Bryant"
[22] "30.5"       "31"          "Basketball"
>
> is.vector(x)
[1] TRUE
```

For reading the data as a list, the `skip` option is used to skip the first line containing the column names. The components of the list are named with the corresponding data type.

```
> x <- scan("athletedata.txt", skip = 1)
+       what = list(Name = character(),
+                   Salary = numeric(),
+                   Endorsements = numeric(),
+                   Sport = character()))
Read 5 records
```

The data is returned as a `list` as shown below.

```
> x
$Name
[1] "Mayweather" "Ronaldo" "James" "Messi"
[5] "Bryant"

$Salary
```

Loading [Contrib]/a11y/accessibility-menu.js

```
1.7 30.5
```

```
$Endorsements
[1] 0 28 53 23 31

$Sport
[1] "Boxing "Soccer" "Basketball" "Soccer"
[5] "Basketball"
```

If the data is desired in the form of the data frame, the list is then converted to a data frame.

```
> as.data.frame(x)
        Name  Salary  Endorsements       Sport
1 Mayweather   105.0             0      Boxing
2    Ronaldo    52.0            28      Soccer
3      James    19.3            53  Basketball
4      Messi    41.8            23      Soccer
5     Bryant    30.5            31  Basketball
```

The `read.table` function provides a convenient mechanism to read tabular data from a file into a data frame. The default value for the `header` argument is `FALSE`. Since the data has the header, the option is set to `TRUE` in the following example.

```
> athlete.info <- read.table("athletedata.txt",
+         header = TRUE)
>
> athlete.info
        Name  Salary  Endorsements       Sport
1 Mayweather   105.0             0      Boxing
2    Ronaldo    52.0            28      Soccer
3      James    19.3            53  Basketball
4      Messi    41.8            23      Soccer
5     Bryant    30.5            31  Basketball
```

The row names for the data frame may be assigned, if needed, after the data is read, or can be directly specified while reading the data as shown below.

```
> athlete.info <- read.table("athletedata.txt",
+   header = TRUE)
+   row.names = c("First", "Second", "Third",
+                 "Fourth" "Fifth")
>
> athlete.info
        Name  Salary  Endorsements       Sport
1 Mayweather   105.0             0      Boxing
2    Ronaldo    52.0            28      Soccer
3      James    19.3            53  Basketball
4      Messi    41.8            23      Soccer
5     Bryant    30.5            31  Basketball
```

The default separator (`sep` argument) is the white space. If any other character delimits the data, the `sep` argument can be set to the delimiter.

For data sets available in CSV (comma-separated values) format, the `read.csv` function is used instead. The default value for the `header` argument is `TRUE`.

```
> athlete.info <- read.csv("athletedata.csv",
+      header=TRUE)
```

Loading [Contrib]/a11y/accessibility-menu.js r reading data files from the internet. The first argument is the URL for the data set.

```
> athlete.info <- read.table(
+    "http://kalathur.com/cs544/data/athletedata.txt",
+     header = TRUE)

> athlete.info <- read.csv(
+    "http://kalathur.com/cs544/data/athletedata.csv",
+     header = TRUE)
```

The data frames can be written to the local file system using the `write.table` or the `write.csv` functions. The row names are written by default, hence using the `FALSE` value for the `row.names` argument suppresses them. If the row names are also written, reading the data back preserves the row names.

The character data is enclosed in double quotes by default. If quotes are not needed, they can be omitted using the `quote` option.

```
> write.table(athlete.info, file="test.txt",
+   row.names = FALSE, quote = FALSE)

> write.csv(athlete.info, file="test.csv",
+   row.names = FALSE, quote = FALSE)
```

The data is written to the current working directory as shown below.

```
> dir()
[1] "athletedata.csv" "athletedata.txt" "test.csv"
[4] "test.txt"
```

Boston University Metropolitan College

Loading [Contrib]/a11y/accessibility-menu.js