Week 6 Module 3: Tables

CSCI E-5a: Programming in R

Let's clear the environment:

```r
rm( list = ls() )
```

Let's load the objects for this module:

```r
load( "Module 3 R Objects.Rdata")

ls()
```

```
## [1] "large.survey.response.character.string.vector"
## [2] "location.character.string.vector"
## [3] "one.week.cereal.brand.character.string.vector"
```

Module Overview and Learning Objectives

Hello! And welcome to Module 3: Tables.

In this module, we'll learn how to summarize categorical data.

- In Section 1, we'll see how to summarize categorical data by using a table of sample frequency counts.
- In Section 2, we'll learn how to summarize categorical data by using a table of sample relative proportions.
- In Section 3, we'll see how to round numeric values.
- In Section 4, we'll learn how to re-order the elements of a table.
- In Section 5, we'll see how to use tables to construct pie charts.

When you've completed this module, you should be able to:

- Summarize categorical data by constructing a table of frequency counts
- Summarize categorical data by constructing a table of relative proportions
- Use the `round()` function to modify numeric values
- Re-order a table
- Use tables with pie charts

There are four new built-in R functions in this module:

- `table()`
- `proportions()`
- `prop.table()`
- `round()`

Section 1: Summarizing sample frequency counts

Main Idea: *We can summarize categorical data by using a table of sample frequency counts*

In this section, we'll see how to summarize categorical data by using a table of sample frequency counts.

How can we summarize a sample of values from a categorical variable?

It doesn't really make sense to calculate a mean or a variance, because categorical data is not numeric data.

Instead, the standard approach is to construct a table of the sample frequency counts.

That is, we count up how many times each category or level occurs in the sample.

For instance, suppose we have 5 transactions of cereal:

Transaction	Brand Name
1	Sugar Bomz
2	Sugar Bomz
3	Healthy Kale and Tofu
4	Sugar Bomz
5	Krispee Yummm!!

Then there were 3 occurrences of Sugar Bomz, 1 occurrence of Krispee Yummm!!, and 1 occurrence of Healthy Kale and Tofu.

Thus, the table of sample frequency counts is:

Brand Name	sample Frequency Count
Sugar Bomz	3
Krispee Yummm!!	1
Healthy Kale and Tofu	1

We can construct tables in R using the `table()` function.

The `table()` function works with both character string and factor representations for categorical data.

Let's start with the character string representation of the cereal sales:

```r
cereal.brand.character.string.vector <-
  c( "Sugar Bomz", "Sugar Bomz", "Healthy Kale and Tofu",
     "Sugar Bomz", "Krispee Yummm!!" )
```

The `table()` function can take a character string vector as its input argument and constructs a table of the sample frequency counts:

```
cereal.brand.frequency.count.table <-
  table(
    cereal.brand.character.string.vector
  )

cereal.brand.frequency.count.table
```

```
## cereal.brand.character.string.vector
## Healthy Kale and Tofu       Krispee Yummm!!            Sugar Bomz
##                     1                     1                     3
```

The table data structure is an actual class in the R object system:

```
class( cereal.brand.frequency.count.table )
```

```
## [1] "table"
```

OK, I admit – with such a simple example, it's hard to get excited about this.

So let's look at a more challenging example, and we'll use the vector `location.character.string.vector` as an example.

Let's start by taking a look at the first few values in this vector:

```
head( location.character.string.vector, n = 8 )
```

```
## [1] "Boston"         "Shanghai"       "Shanghai"       "Shanghai"
## [5] "Boston"         "Shanghai"       "Salt Lake City" "Shanghai"
```

In fact, there are four levels for the categorical data in this vector:

```
unique( location.character.string.vector )
```

```
## [1] "Boston"         "Shanghai"       "Salt Lake City" "London"
```

There are also a lot of elements in this vector, so we probably don't want to compile the table by hand:

```
length( location.character.string.vector )
```

```
## [1] 235
```

Even though there are many more elements in `location.character.string.vector` than in `cereal.brand.character.stri`
it's no more difficult to construct a table of the sample frequency counts:

```
location.frequency.count.table <-
  table(
    location.character.string.vector
  )

location.frequency.count.table
```

```
## location.character.string.vector
##         Boston         London Salt Lake City       Shanghai
##             64             26             47             98
```

The `table()` function can also take a factor as its input argument.

Let's construct the factor `location.factor`:

```
location.factor <-
  factor( location.character.string.vector )
```

Here are the first few values of `location.factor`:

```
head( location.factor, n = 8 )
```

```
## [1] Boston         Shanghai       Shanghai       Shanghai       Boston
## [6] Shanghai       Salt Lake City Shanghai
## Levels: Boston London Salt Lake City Shanghai
```

Remember that when we directly display a factor, the levels are not enclosed in quotes, and R reports all the pre-specified levels for the factor, even if they don't appear explicitly in the output.

Now let's make a table of the sample frequency counts for this factor:

```
table( location.factor )
```

```
## location.factor
##         Boston         London Salt Lake City       Shanghai
##             64             26             47             98
```

So that's how we can summarize categorical data by using a table of frequency counts.

Now let's see how to summarize categorical data by using a table for sample relative proportions.

Exercise 1: Constructing a frequency count table

The vector `one.week.cereal.brand.character.string.vector` contains data on cereal brands sold during one week.

First, construct a frequency count table for the cereal brands using this character string vector, save it in a variable, and display it directly.

Next, create a factor from this character string variable, and save it in a variable.

Then use this factor to construct a new table of the sample frequency count frequencies for this data.

Solution

Section 2: Summarizing sample relative proportions

Main Idea: *We can summarize categorical data by using a table of sample relative proportions*

In this section, we'll see how to summarize categorical data by using a table of sample relative proportions.

Often, we aren't interested in the actual frequency counts in a sample, but rather the sample relative proportions.

For instance, recall the frequency count table for cereal sales:

Brand Name	sample Frequency Count
Sugar Bomz	3
Krispee Yummm!!	1
Healthy Kale and Tofu	1

There were 3 sales of Sugar Bomz out of a total of 5 transactions, so the sample relative proportion of sales for Sugar Bomz is $3/5 = 0.60$.

Thus, we have:

Brand Name	sample Relative Proportion
Sugar Bomz	0.6
Krispee Yummm!!	0.2
Healthy Kale and Tofu	0.2

To construct a table of relative proportions in R, we can use the `proportions()` function.

The `proportions()` function takes a sample frequency count table that's been created by the `table()` function, and returns a new table consisting of the sample relative proportions across the levels:

```
proportions(
  cereal.brand.frequency.count.table
)
```

```
## cereal.brand.character.string.vector
## Healthy Kale and Tofu       Krispee Yummm!!            Sugar Bomz
##                   0.2                   0.2                   0.6
```

Let's try this with the location data:

```
location.relative.proportions.table <-
  proportions( location.frequency.count.table )

location.relative.proportions.table
```

```
## location.character.string.vector
##         Boston         London Salt Lake City       Shanghai
##      0.2723404      0.1106383      0.2000000      0.4170213
```

Remember!!: In order to construct a table of the sample relative proportions, you first have to create a table of the sample frequency counts table, and then pass this to the `proportions()` function.

This function is also called `prop.table()`:

```
prop.table( location.frequency.count.table )
```

```
## location.character.string.vector
##          Boston          London Salt Lake City       Shanghai
##       0.2723404       0.1106383      0.2000000      0.4170213
```

So that's how to summarize categorical data by using a table of sample relative proportions.

Now let's learn about the `round()` function.

Exercise 2: Constructing a sample relative proportions table

Construct a sample relative proportions table for the cereal brands using the frequency count table you created in Exercise 1. Save this table in a variable, and display it directly.

Solution

Section 3: Rounding Numeric Values

> **Main Idea:** *We can round numeric values*

In this section, we'll see how to round numeric values.

You might have noticed that the entries in the table of relative proportions that we just constructed have a lot of decimal places.

Is there a way that we can clean this up, so that each entry is displayed with exactly 2 decimal places?

One approach is to use the `formatC()` function:

```
formatC(
  x = location.relative.proportions.table,
  format = "f",
  digits = 2
)
```

```
## location.character.string.vector
##          Boston          London Salt Lake City       Shanghai
##          "0.27"          "0.11"         "0.20"         "0.42"
```

This isn't bad, but the quotes around the numerical values are annoying.

Instead, we can use the `round()` function, which takes two input arguments:

- A numeric value, denoted x.
- The number of digits to round to, denoted *digits*.

The `round()` function then returns a new numeric value, which is the input argument x rounded to the specified number of digits.

If we use the `round()` function with a table of sample relative proportions, the entries are displayed without the enclosing quotes:

```
round(
  location.relative.proportions.table,
  digits = 2
)
```

```
## location.character.string.vector
##          Boston          London Salt Lake City       Shanghai
##            0.27            0.11           0.20           0.42
```

It's important to understand the difference between `formatC()` and `round()`:

- The `formatC()` function takes a numeric value as an input argument, and returns a character string value which is the formatted numeric value.
- The `round()` function also takes a numeric value as an input argument, but it returns another numeric value.

Thus, you can do arithmetic with the output of the `round()` function, because it's a numeric value, but you can't do arithmetic with the output of the `formatC()` function, because it's a character string value.

Also, note that when you format a numeric value using `formatC()` the output is guaranteed to have the specified number of decimal places, but this is not the case with the `round()` function.

For instance, if we want to format the value 0.2 with 3 decimal places, we have:

```
formatC(
  0.2,
  format = "f",
  digits = 3
)
```

```
## [1] "0.200"
```

We'll always have 3 digits when use the `formatC()` function like this.

However, with the `round()` function it's a little different.

When R displays numeric values it does not print trailing 0 characters, so even if we try to use the `round()` function we won't get exactly 3 decimal places:

```
round(
  0.30,
  digits = 3
)
```

```
## [1] 0.3
```

We've seen many situations where it's important to display numeric values with a specific number of decimal places, so for those situations the `round()` function is undesirable because the decimal expansion will be truncated.

On the other hand, the `round()` function can often be very useful, and it doesn't result in a display with quotes.

In the end, you always have to use your judgment and insight when determining the best way to format your data.

So that's how to round numeric values by using the `round()` function.

Now let's see how to re-order the elements of a table.

Exercise 3: Rounding the relative proportions table numbers

Use the `round()` function so that the numbers displayed in the relative proportions table have 3 decimal places.

Solution

Section 4: Re-Ordering a Table

> **Main Idea:** *We can re-order the entries in a table*

In this section, we'll learn how to re-order the elements of a table.

There are three different methods that we can use to control the order in which table elements are displayed:

- Using character string indexing on the table names.
- Pre-specifying the levels for a factor.
- Sorting the table numerically.

Notice in all of the examples we've seen so far that we haven't controlled the order that the table displays the different categories or levels.

One way to control the display order is to use character string indexing on the table names.

When we construct a table from a character string representation of categorical data, R automatically orders the categories alphabetically.

For instance, let's display the table for the frequency counts for cereal brand sales:

```
cereal.brand.frequency.count.table
```

```
## cereal.brand.character.string.vector
## Healthy Kale and Tofu       Krispee Yummm!!            Sugar Bomz
##                     1                     1                     3
```

Notice that the table entries are listed alphabetically.

If we want to list the table entries in a different sequence, we can use a character string vector with the table names to create a new table with the entries displayed in the desired order.

For instance, let's suppose we wanted to list Sugar Bomz first, then Krispee Yummm!!, and finally Healthy Kale and Tofu.

Let's first create a character string vector with these brand names in the desired order:

```
reordering.brand.name.character.string.vector <-
  c(
    "Sugar Bomz",
    "Krispee Yummm!!",
    "Healthy Kale and Tofu"
  )
```

Now we can reorder the table using character string indexing:

```
cereal.brand.frequency.count.table[
  reordering.brand.name.character.string.vector
]
```

```
## cereal.brand.character.string.vector
##            Sugar Bomz       Krispee Yummm!! Healthy Kale and Tofu
##                     3                     1                     1
```

Essentially, the table is operating like a named vector, where the levels are the names of the vector.

Remember!! All the brand names are stored into 1 character vector, and then we use this vector for the character string indexing.

In fact, we don't even have to do all that – we can just construct the vector of character string names inside the square brackets:

```
cereal.brand.frequency.count.table[
  c( "Sugar Bomz", "Krispee Yummm!!", "Healthy Kale and Tofu" )
]
```

```
## cereal.brand.character.string.vector
##            Sugar Bomz       Krispee Yummm!! Healthy Kale and Tofu
##                     3                     1                     1
```

This is very compressed, and you don't have to write code like this.

Let's try this with the location data.

Suppose that we want to list the cities in this order: Shanghai, London, Boston, Salt Lake City.

Then we could do this:

```
location.frequency.count.table[
  c( "Shanghai", "London", "Boston", "Salt Lake City" )
]
```

```
## location.character.string.vector
##       Shanghai         London         Boston Salt Lake City
##             98             26             64             47
```

Another way to control the ordering of the levels in a table is to specify the levels when creating a factor.

If we construct a table from a factor, then the order of the elements of the table will follow the sequence of levels of the factor.

If we don't specify a vector of possible levels when we create the factor, then R will order the levels alphabetically, and we are back to the situation with character string vectors.

On the other hand, if we *do* specify the levels when we create the factor, then R will use the order of the levels when displaying the table.

Let's reconstruct the `location.factor`, but with a different sequence for the levels:

```
reordered.location.factor <-
  factor(
    x = location.character.string.vector,
    levels = c("Shanghai", "London", "Boston", "Salt Lake City" )
  )
```

Now when we construct the table, the levels will automatically be displayed in the correct order:

```
table( reordered.location.factor )
```

```
## reordered.location.factor
##       Shanghai         London         Boston Salt Lake City 
##             98             26             64             47
```

This is one of the advantages of using factors: by pre-specifying the levels once when the factor is created, we can then just use the factor and it will always be displayed in the desired order without any further fiddling on our part.

You might think that the whole issue of the ordering the display of the levels is not very important, but it actually comes up a lot in practice.

We often work with categorical data in which there is some intrinsic ordering of levels that is not alphabetical.

For instance, consider a customer satisfaction survey at a restaurant where customers are presented with the statement "I enjoyed my dining experience" and must respond using one of 5 responses:

- "Strongly Disagree"
- "Disagree"
- "No Opinion"
- "Agree"
- "Strongly Agree"

There is a natural ordering here, going from "Strongly Disagree" to "Strongly Agree".

Now suppose we have this data:

Survey	Response
1	Strongly Agree
2	Disagree
3	Agree
4	No Opinion
5	Agree

We can represent this data using a character string vector:

```
survey.response.character.string.vector <-
  c(
    "Strongly Agree",
    "Disagree",
    "Agree",
    "No Opinion",
    "Agree"
  )
```

Now let's create a sample frequency count table of this data:

```
survey.response.frequency.count.table <-
  table(
    survey.response.character.string.vector
  )

survey.response.frequency.count.table
```

```
## survey.response.character.string.vector
##          Agree       Disagree     No Opinion Strongly Agree
##              2              1              1              1
```

Notice how weird this is: the table starts with the level "Agree", then the next level is "Disagree", then "No Opinion", and finally "Strongly Agree".

Instead of re-ordering the table by hand, we can instead create a factor with the proper ordering of the levels:

```
survey.response.levels.vector <-
  c(
    "Strongly Disagree",
    "Disagree",
    "No Opinion",
    "Agree",
    "Strongly Agree"
  )

survey.response.factor <-
  factor(
    survey.response.character.string.vector,
    levels = survey.response.levels.vector,
  )

survey.response.factor
```

```
## [1] Strongly Agree Disagree       Agree          No Opinion     Agree
## Levels: Strongly Disagree Disagree No Opinion Agree Strongly Agree
```

When we use this factor the table will display with the levels in the proper order:

```
table(
  survey.response.factor
)
```

```
## survey.response.factor
## Strongly Disagree          Disagree        No Opinion             Agree
##                 0                 1                 1                 2
##    Strongly Agree
##                 1
```

Notice that the table includes an entry for the "Strongly Disagree" level, even though this did not occur in our data.

This was possible because we pre-specified the levels when creating the factor.

It's not possible to do this if we just pass a character string vector to the `table()` function – after all, the only way that it can construct the levels is by using the values in the vector itself, so if a level is possible but doesn't doesn't actually occur in the data then there's no way for R to know about it.

This is another advantage to working with factors over character strings: it enables us to include levels in tables even if they don't occur in the observed data.

Finally, there is one more way that we can re-order the display of the elements in a table.

Often, we would like the table to be sorted in terms of the numerical values themselves.

To achieve this, we can sort the table using the `sort()` function:

```
sort( location.frequency.count.table )
```

```
## location.character.string.vector
##          London Salt Lake City         Boston       Shanghai
##              26             47             64             98
```

We can also sort in decreasing order:

```
decreasing.location.frequency.count.table <-
    sort(
      x = location.frequency.count.table,
      decreasing = TRUE
    )

decreasing.location.frequency.count.table
```

```
## location.character.string.vector
##        Shanghai         Boston Salt Lake City         London
##              98             64             47             26
```

So that's how to re-order the entries in a table.

Now let's see how to use tables to construct pie charts.

Exercise 4: Re-ordering the levels

Using character string indexing on the levels of the frequency count table that you created in Exercise 1 to create a new frequency count table displaying the brands in the order "Krispee Yummm!!", "Healthy Kale and Tofu", and "Sugar Bomz".

Then create a new factor with the levels pre-specified in the order "Krispee Yummm!!", "Healthy Kale and Tofu", and "Sugar Bomz". Save this factor in a variable, and use it to create another frequency count table, again displaying the brand names in this order.

Finally, sort the table in *ascending order*, so that the brand with the least sales is reported first, and the brand with the most sales is reported last.

Solution

Section 5: Tables and Pie Charts

Main Idea: *We can use tables to construct pie charts*

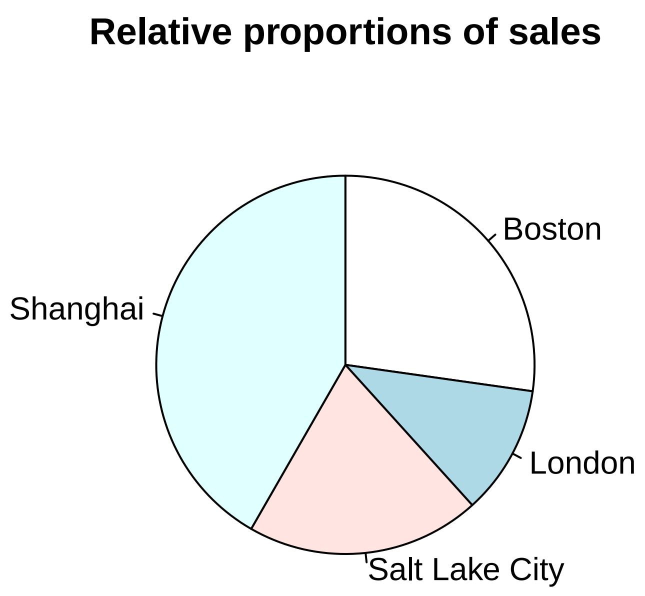
In this section, we'll see how to use tables to construct pie charts.

Tables play nicely with pie charts.

In fact, once we've created a table, we can pass it directly to the `pie()` function.

The labels for the pie charts will come directly from the table.

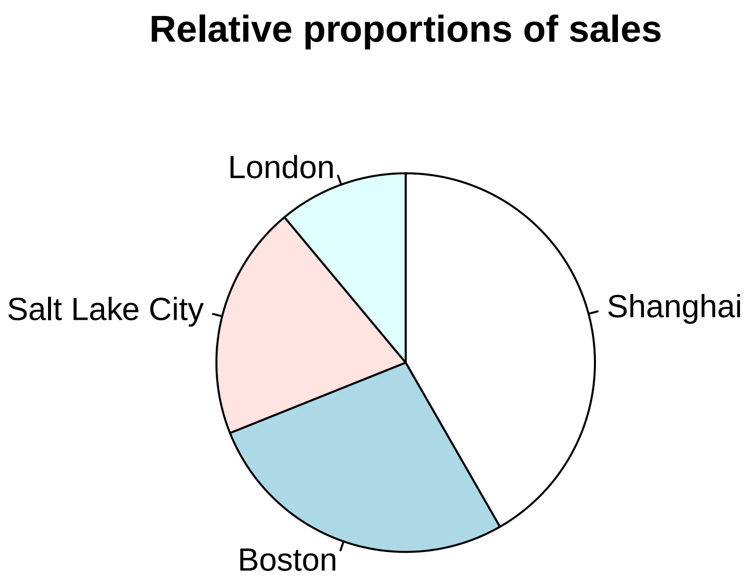
```r
pie(
    location.frequency.count.table,
    main = "Relative proportions of sales",
    clockwise = TRUE
)
```



It actually doesn't matter whether or not we use a frequency count table, or a relative proportions table, because the pie chart display will effectively convert everything into a relative proportions anyway.

Sorting the table in decreasing order can give us a pie chart where the pie slices are in decreasing order:

```r
pie(
    decreasing.location.frequency.count.table,
    main = "Relative proportions of sales",
    clockwise = TRUE
)
```



So that's how to use tables to construct pie charts.

Now let's review what we've learned in this module.

Exercise 5: Table and Pie Charts

Construct a pie chart of the relative proportions of the cereal brands using the frequency count table you constructed in Problem 1.

Then construct a pie chart of the relative proportions of the cereal brands using the frequency count table you constructed in Problem 1, displayed in descending order.

Solution

Module Review

In this module, we learned how to summarize categorical data.

- In Section 1, we saw how to summarize categorical data by using a table of sample frequency counts.
- In Section 2, we learned how to summarize categorical data by using a table of sample relative proportions.
- In Section 3, we saw how to round numeric values.
- In Section 4, we learned how to re-order the elements of a table.

- In Section 5, we saw how to use tables to construct pie charts.

Now that you've completed this module, you should be able to:

- Summarize categorical data by constructing a table of frequency counts using the `table()` function.
- Summarize categorical data by constructing a table of relative proportions using the `table()` function.
- Use the `round()` function to modify numeric values.
- Re-order a table.
- Use tables with pie charts.

There were four new built-in R functions in this module:

- `table()`
- `proportions()`
- `prop.table()`
- `round()`

Solutions to the Exercises

Exercise 1: Constructing a frequency count table

The vector `one.week.cereal.brand.character.string.vector` contains data on cereal brands sold during one week.

First, construct a frequency count table for the cereal brands using this character string vector, save it in a variable, and display it directly.

Next, create a factor from this character string variable, and save it in a variable.

Then use this factor to construct a new table of the sample frequency count frequencies for this data.

Solution

Here's the table constructed from the character string vector:

```
one.week.cereal.brand.frequency.count.table <-
  table(
    one.week.cereal.brand.character.string.vector
    )

one.week.cereal.brand.frequency.count.table
```

```
## one.week.cereal.brand.character.string.vector
## Healthy Kale and Tofu       Krispee Yummm!!            Sugar Bomz
##                    41                   117                   173
```

Next we can create the factor:

```
one.week.cereal.brand.factor <-
  factor(
    one.week.cereal.brand.character.string.vector
  )
```

And finally we can construct another frequency count table:

```
one.week.cereal.brand.frequency.count.table <-
  table(
    one.week.cereal.brand.factor
  )

one.week.cereal.brand.frequency.count.table
```

```
## one.week.cereal.brand.factor
## Healthy Kale and Tofu       Krispee Yummm!!            Sugar Bomz
##                    41                   117                   173
```

Exercise 2: Constructing a sample relative proportions table

Construct a sample relative proportions table for the cereal brands using the frequency count table you created in Exercise 1. Save this table in a variable, and display it directly.

Solution

```
one.week.cereal.brand.relative.proportions.table <-
  proportions(
    one.week.cereal.brand.frequency.count.table
  )

one.week.cereal.brand.relative.proportions.table
```

```
## one.week.cereal.brand.factor
## Healthy Kale and Tofu       Krispee Yummm!!            Sugar Bomz
##             0.1238671             0.3534743             0.5226586
```

Exercise 3: Rounding the relative proportions table numbers

Use the `round()` function so that the numbers displayed in the relative proportions table have 3 decimal places.

Solution

```
round(
  one.week.cereal.brand.relative.proportions.table,
  digits = 3
)
```

```
## one.week.cereal.brand.factor
## Healthy Kale and Tofu       Krispee Yummm!!            Sugar Bomz
##                 0.124                 0.353                 0.523
```

Exercise 4: Re-ordering the levels

Using character string indexing on the levels of the frequency count table that you created in Exercise 1 to create a new frequency count table displaying the brands in the order “Krispee Yummm!!”, “Healthy Kale and Tofu”, and “Sugar Bomz”.

Then create a new factor with the levels pre-specified in the order “Krispee Yummm!!”, “Healthy Kale and Tofu”, and “Sugar Bomz”. Save this factor in a variable, and use it to create another frequency count table, again displaying the brand names in this order.

Finally, sort the table in *ascending order*, so that the brand with the least sales is reported first, and the brand with the most sales is reported last.

Solution

```
reordered.one.week.cereal.brand.frequency.count.table <-
  one.week.cereal.brand.frequency.count.table[
    c(
      "Krispee Yummm!!",
      "Healthy Kale and Tofu",
      "Sugar Bomz"
    )
  ]
```

```
reordered.one.week.cereal.brand.factor <-
  factor(
    one.week.cereal.brand.character.string.vector,
    levels =
      c(
        "Krispee Yummm!!",
        "Healthy Kale and Tofu",
        "Sugar Bomz"
      )
  )

table(
  reordered.one.week.cereal.brand.factor
)
```

```
## reordered.one.week.cereal.brand.factor
##       Krispee Yummm!! Healthy Kale and Tofu            Sugar Bomz
##                   117                    41                   173
```

```
ascending.order.one.week.cereal.brand.frequency.count.table <-
  sort( one.week.cereal.brand.frequency.count.table )

ascending.order.one.week.cereal.brand.frequency.count.table
```

```
## one.week.cereal.brand.factor
## Healthy Kale and Tofu       Krispee Yummm!!            Sugar Bomz
##                    41                   117                   173
```

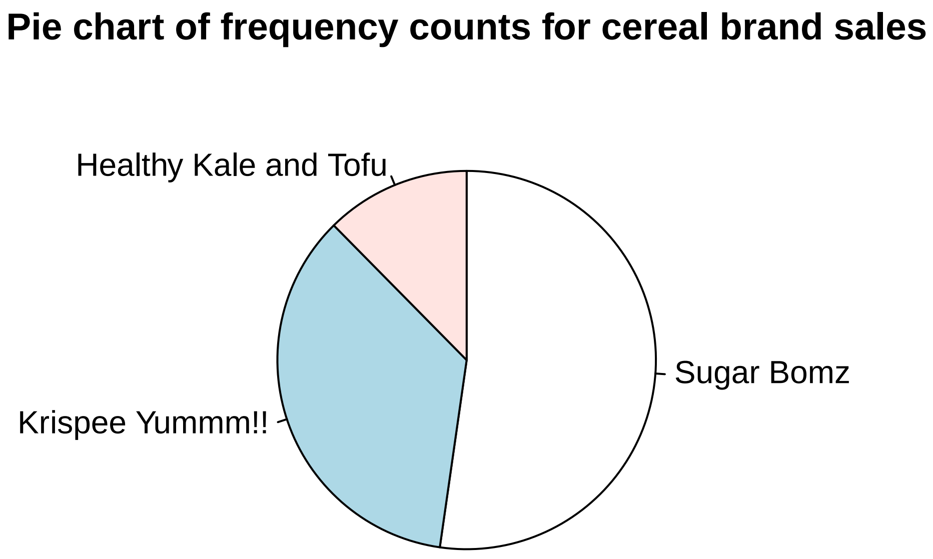
Exercise 5

Construct a pie chart of the relative proportions of the cereal brands using the frequency count table you constructed in Problem 1.

Then construct a pie chart of the relative proportions of the cereal brands using the frequency count table you constructed in Problem 1, displayed in decreasing order.

Solution

```
pie(
  sort(
    one.week.cereal.brand.frequency.count.table,
    decreasing = TRUE
  ),
  main = "Pie chart of frequency counts for cereal brand sales",
  clockwise = TRUE
)
```


Pie chart of frequency counts for cereal brand sales
Healthy Kale and Tofu
Sugar Bomz
Krispee Yummm!!