Week 6 Module 2: Factors

CSCI E-5a: Programming in R

Let's clear the global computing environment:

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

Module Overview and Learning Objectives

Hello! And welcome to **Module 2: Factors**.

In this module, we'll learn how to work with factors.

- In Section 1, we'll see how to specify the levels of a factor.
- In Section 2, we'll learn how to specify the labels of a factor.
- In Section 3, we'll see how to re-order the levels.
- In Section 4, we'll learn how to group levels together.
- In Section 5, we'll investigate the `cut()` function.

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

- Specify the levels of a factor.
- Specify the labels of a factor.
- Re-order the levels of a factor.
- Group factor levels together.
- Create a factor from a numeric vector.

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

- `levels()`
- `cut()`

All right, let's get started by learning how to specify the levels of a factor.

Section 1: Specifying the Levels

Main Idea: *We can specify the levels of a factor*

In this section, we'll see how to specify the levels of a factor.

In the previous lecture, I mentioned that factors use less space than the equivalent character string representation, and operations on factors execute more quickly.

While this is true, in practice the difference between factors and string representations in terms of improved efficiency is usually not important unless you're working on a truly gigantic project.

Instead, I think the real power of factors comes from the fact that we can specify the levels of the categorical variable when we create the factor, and this information is stored in the factor data structure.

Then, whenever we use the factor, R can refer to the stored information about the levels.

If you're working on a large project, and are doing a lot of analyses, being able to store a properly structured set of levels can be a real time-saver.

So how do we specify the structure of the levels?

So far, we've created a factor by simply calling the `factor()` function with just one input argument, the vector of observed values.

When we do this, R automatically determines the levels by scanning the observed values and compiling a vector of unique representatives, which then become the levels of the factor if we don't do anything.

But we can also force R to use a specific set of values for the levels by using the `levels` option.

Specifying the levels in this manner is appealing because it enables us to exactly enforce the allowable levels.

Let's look at a simple example to see how this works.

We'll start with a simple character string vector:

```
cereal.brand.character.string.vector <-
    c("sbz", "kym", "sbz",
      "xyz", "kym", "sbz" )
```

Now we can use this simple character vector to construct a factor:

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

cereal.brand.factor
```

```
## [1] sbz kym sbz xyz kym sbz
## Levels: kym sbz xyz
```

Let's review the levels for this factor:

```
levels( cereal.brand.factor )
```

```
## [1] "kym" "sbz" "xyz"
```

Notice that one of the values in the character vector is "xyz", and clearly this is a mistake.

If we just pass the character string vector to the `factor()` function with no extra information, R will include "xyz" in the resulting factor levels, because there is no way for it to determine that this is an invalid value.

However, if we specify the levels by using the `levels` option, then R will generate an `NA` for any values not in the specified vector of allowable levels:

```
corrected.cereal.brand.factor <-
    factor(
        x = cereal.brand.character.string.vector,
        levels = c("sbz", "kym")
    )

corrected.cereal.brand.factor
```

```
## [1] sbz  kym  sbz  <NA> kym  sbz
## Levels: sbz kym
```

Essentially, by specifying the levels with a character vector we are effectively telling R, 'The label `"sbz"` should be level 1, the label "'kym" should be level 2, . . .' and so on.

Then R performs simple pattern matching to each element of the x data vector, and if it finds a match then it assigns the appropriate numeric code to the corresponding value in the factor.

Notice that "xyz" is *not* included among the specified levels, and this is converted to an `NA` value.

Now we can find out the location of the invalid value:

```
which(
  is.na(
    corrected.cereal.brand.factor
  )
)
```

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

This is a very useful feature when cleaning data, because we often know in advance what the levels are supposed to be and so we can pre-specify them without knowing anything about the actual observed values, and this allows us to detect mistakes.

Specifying the levels when creating a factor also enables us to include levels that aren't actually observed in the data.

When R constructs the levels for a factor without a specified set of levels, the only thing it can do is to use the set of observed levels in the data.

Thus, if a particular value does not occur in the observed data, it won't be included in the set of levels.

For instance, you might have noticed that the character string vector did not actually contain an entry for Healthy Kale and Tofu.

Thus, when we use the `factor()` function to create a factor from this vector, there won't be a level for Healthy Kale and Tofu.

However, we can include an entry for Healthy Kale and Tofu when we specify the levels, and this will be stored with the other levels, even though it was not realized in the observed data:

```
complete.cereal.brand.factor <-
    factor(
        cereal.brand.character.string.vector,
        levels = c("sbz", "kym", "hkt")
    )

complete.cereal.brand.factor
```

```
## [1] sbz  kym  sbz  <NA> kym  sbz
## Levels: sbz kym hkt
```

Notice that while the observed data is still the same as before, now the displayed levels includes “hkt”, even though it wasn’t observed in the dataset.

Thus, specifying the levels of a factor when creating it allows us to:

- Exclude values that are observed but are not valid level labels.
- Include values that are valid level labels but are not observed.

So that’s how we specify the levels of a factor.

Now let’s see how to specify the labels of a factor.

Exercise 2.1: Specifying the levels

Suppose we have a very simple vector of character string labels:

```
simple.character.string.vector <-
  c( "blue", "blue", "green", "red", "green", "blue", "red" )
```

Construct a factor that consists of just the values “green” and “blue”, and uses the value `NA` when a “red” value occurs in the original character vector.

Solution

Section 2: Specifying the Labels

> **Main Idea:** *We can specify the labels of a factor*

In this section, we’ll learn how to specify the labels of a factor.

What happens if we have a factor, and then decide that we don’t like the labels that are used for the levels?

Currently, the labels for the cereal brands are all lower-case:

```
levels( complete.cereal.brand.factor )
```

```
## [1] "sbz" "kym" "hkt"
```

Our usual convention is that the cereal brand values are “SBZ”, “KYM”, and “HKT”, instead of “sbz”, “kym”, “hkt”.

We can destructively modify the labels for the levels by directly assigning a character string vector to the factor labels, which are returned by the `levels()` function:

```
levels( complete.cereal.brand.factor ) <-
    c( "SBZ", "KYM", "HKT" )

complete.cereal.brand.factor
```

```
## [1] SBZ  KYM  SBZ  <NA> KYM  SBZ
## Levels: SBZ KYM HKT
```

The `levels()` function is on the left-hand side of an assignment operation, so that means that we are *destructively modifying* the levels of `complete.cereal.brand.factor`.

We’re not creating a new factor – we’re altering the internal structure of the R object.

Notice that when we changed the levels we retained the original order.

In fact, if we re-assign the levels and don’t keep the original order, then we’ll corrupt the data in the factor.

By re-ordering the levels we are actually changing the association of the levels with the numeric representation, but the vector with the numeric values remains the same.

To see this, let’s first make a copy of `complete.cereal.brand.factor`:

```
bad.cereal.brand.factor <-
    complete.cereal.brand.factor

bad.cereal.brand.factor
```

```
## [1] SBZ  KYM  SBZ  <NA> KYM  SBZ
## Levels: SBZ KYM HKT
```

Here is the vector of numeric representatives:

```
as.numeric( bad.cereal.brand.factor )
```

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

Now let’s reassign the levels using a different order:

```
levels( bad.cereal.brand.factor ) <-
    c( "HKT", "KYM", "SBZ" )

bad.cereal.brand.factor
```

```
## [1] HKT  KYM  HKT  <NA> KYM  HKT
## Levels: HKT KYM SBZ
```

And now we have corrupted the original data: for instance, the entries that originally represented Sugar Bomz now contain the label for Healthy Kale and Tofu.

Notice that we haven’t changed the numeric representation:

```
as.numeric( bad.cereal.brand.factor )
```

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

So this is still the same as before.

In fact, that's the whole problem here: we've changed the association between the levels and their numeric representation, but R doesn't make the corresponding change to the numeric vector.

So, the moral of the story is that if you want to change the labels for the levels by directly assigning a character string vector, then you must preserve the ordering of the levels.

In fact, we can actually reassign the labels when we originally create the factor by using the `labels` option.

By assigning a character string vector to the `labels` option, R will replace the original labels observed in the data with the new values.

For instance, let's recall the original data of character string values:

```
cereal.brand.character.string.vector
```

```
## [1] "sbz" "kym" "sbz" "xyz" "kym" "sbz"
```

Now we'll construct a factor using both the `levels` and `labels` options:

```
final.cereal.brand.factor <-
  factor(
    x = cereal.brand.character.string.vector,
    levels = c( "sbz", "kym", "hkt" ),
    labels = c( "SBZ", "KYM", "HKT" )
  )

final.cereal.brand.factor
```

```
## [1] SBZ  KYM  SBZ  <NA> KYM  SBZ
## Levels: SBZ KYM HKT
```

We did a lot in this one call to `factor()`:

- First, by specifying the valid levels we excluded the invalid value "xyz".
- Second, by specifying the valid levels we included the valid value "hkt" among the levels, even though it was not actually observed in the data.
- Finally, we replaced the lower-case labels with upper-case labels using the `labels` option.

So that's how to specify the labels of a factor.

Now let's see how to re-order the levels of a factor.

Exercise 2.2: Changing the Levels

Using the vector `simple.character.string.vector`, construct a factor with two levels, which have the labels "Green" and "Blue".

Specify the levels using the `levels` option, and specify the labels for the levels using the `labels` option.

Solution

Section 3: Re-Ordering the Levels

> **Main Idea:** *We can re-order the levels of a factor*

In this section, we'll see how to re-order the levels of a factor.

Usually, the actual ordering of the factor levels won't matter, and you don't have to worry about it.

The nice thing is that R manages all the internal details, so it's easy for you to work with factor.

However, sometimes it *does* matter, and you want the levels to be in a specific order.

Let's review how R automatically constructs the levels when constructing a factor from a vector by using the `factor()` function.

First, R compiles a set of unique representatives of the values in the input vector.

These unique representatives are sorted alphabetically, and then are assigned numeric codes in order starting at 1.

For instance, let's recreate the simple version of the cereal brand factor:

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

cereal.brand.factor
```

```
## [1] sbz kym sbz xyz kym sbz
## Levels: kym sbz xyz
```

There are 3 unique values in this vector: "sbz", "kym", and "xyz".

So when this vector is converted to a factor these become the levels of the factor, in alphabetical order:

```
factor( cereal.brand.factor )
```

```
## [1] sbz kym sbz xyz kym sbz
## Levels: kym sbz xyz
```

You can see here that the levels have been sorted alphabetically.

Thus, "kym" has the numeric code 1, "sbz" has the numeric code 2, and "xyz" has the numeric code 3.

We can see this by looking at the numeric representatives of the factor:

```
as.numeric( cereal.brand.factor )
```

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

We normally don't care about the integer representation, and the order of the factor levels is of no interest.

But sometimes we *do* care, because the factor levels might have some sort of intrinsic ordering.

For instance, suppose we have some survey data, where subjects are asked to rate their response to a question on an ordered scale: "Strongly Disagree", "Disagree", "No Opinion", "Agree", or "Strongly Agree".

Here's an example character vector:

```r
agree.disagree.character.string.vector <-
    c(
        "Strongly Disagree", "Strongly Agree",
        "No Opinion", "Agree",
        "Strongly Disagree", "Disagree",
        "No Opinion", "No Opinion",
        "Agree"
    )
```

Let's construct a factor and then examine the levels:

```r
agree.disagree.factor <-
    factor( agree.disagree.character.string.vector )

levels( agree.disagree.factor )
```

```
## [1] "Agree"             "Disagree"          "No Opinion"
## [4] "Strongly Agree"    "Strongly Disagree"
```

You can see that R has alphabetically sorted the levels, and this destroys the natural progression.

This looks weird and makes it difficult to think about the data, so it would be desirable to be able to explicitly force a preferred ordering.

We can do this by constructing a vector of the levels and then instructing R to use this vector when creating the factor:

```r
ordered.agree.disagree.levels <-
    c(
        "Strongly Disagree", "Disagree",
        "No Opinion",
        "Agree", "Strongly Agree"
    )

ordered.agree.disagree.factor <-
    factor(
        agree.disagree.factor,
        levels =
            ordered.agree.disagree.levels
    )

ordered.agree.disagree.factor
```

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

Now let's look at the levels of the re-ordered factor:

```r
levels( ordered.agree.disagree.factor )
```

```
## [1] "Strongly Disagree" "Disagree"          "No Opinion"
## [4] "Agree"             "Strongly Agree"
```

This is much better!

So that's how to re-order the levels of a factor.

Now let's see how to group levels together.

Exercise 2.3: Re-ordering the levels

Re-order the levels of `simple.character.factor` so that "Green" is the first level and "Blue" is the second level.

Solution

Section 4: Grouping Levels

> **Main Idea:** *We can group the levels of a factor together*

In this section, we'll learn how to group the levels of a factor together.

Sometimes we don't want to work with all the factor levels that are in our data.

In other words, our factor data has too much resolution, and we would like to group some of the levels together.

Let's go back to our `final.cereal.brand.factor`:

```
final.cereal.brand.factor
```

```
## [1] SBZ  KYM  SBZ  <NA> KYM  SBZ
## Levels: SBZ KYM HKT
```

Let's group this into two categories:

- Sugar Bomz is classified as a "Value" brand.
- Krispee Yummm!! and Healthy Kale and Tofu are classified as "Premium" brands.

To do this, we assign a new vector to the levels:

```
grouped.cereal.brand.factor <-
  final.cereal.brand.factor

levels( grouped.cereal.brand.factor ) <-
    c( "Value", "Premium", "Premium" )

grouped.cereal.brand.factor
```

```
## [1] Value   Premium Value   <NA>    Premium Value
## Levels: Value Premium
```

Notice that we've now grouped "KYM" and "HKT" together into the "Premium" category.

Be aware that once you've grouped factor levels like this, the original levels can *not* be reconstructed.

You'll notice that I copied the `final.cereal.brand.factor` to a new variable `grouped.cereal.brand.factor` before I grouped the levels.

That way, I haven't lost the information in `final.cereal.brand.factor`.

Grouping factor levels is important for data analysis, because we frequently want to aggregate categories in our data.

It's also extremely useful when cleaning data, because every typo will generate a new category.

Thus, we can repair incorrect data very easily by simply grouping factor levels.

We can actually perform this grouping operation when we first construct the factor with the `factor()` function by using the `labels` option:

```
grouped.cereal.brand.factor <-
  factor(
    x = cereal.brand.character.string.vector,
    levels = c( "sbz", "kym", "hkt" ),
    labels = c( "Value", "Premium", "Premium" )
  )

grouped.cereal.brand.factor
```

```
## [1] Value   Premium Value   <NA>    Premium Value
## Levels: Value Premium
```

There's a lot going on in this code!

- R uses the character strings in the vector for the `levels` option to perform pattern matching on the `cereal.brand.character.string.vector`, and it uses this to determine the numeric codes for the levels.
- The vector for the `levels` option also determines the ordering of the levels, and will enable R to determine which items in the data are invalid.
- The `labels` option then allows us to replace the character string representatives in the data with character strings that we select.
- Finally, duplicate values in the `labels` option will result in R grouping the levels together.

For a short bit of code, this is extremely powerful!

This is the best practice: specify the levels and labels when the factor is being created, as well as performing any grouping operations.

So that's how to group the levels of a factor together.

Now let's see how to convert a numeric vector to a factor.

Exercise 2.4: Grouping levels

Suppose we have a sequence of cereal box sales stored in a vector:

```r
cereal.sales.vector <-
    c( "sbz", "sbz", "kym", "hkt",
       "sbz", "kym", "sb", "ky" )

cereal.sales.factor <-
    factor( cereal.sales.vector )

levels( cereal.sales.factor )
```

```
## [1] "hkt" "ky"  "kym" "sb"  "sbz"
```

We want to represent these sales using our standard 3-letter abbreviation.

Notice that the last 2 entries only use 2 letters for the abbreviation, and R will treat these as distinct levels:

```r
cereal.sales.factor
```

```
## [1] sbz sbz kym hkt sbz kym sb  ky
## Levels: hkt ky kym sb sbz
```

Group the levels "sbz" and "sb" together as "sbz". Likewise, group the levels "kym" and "ky" together as "kym".

Solution

Section 5: The `cut()` Function

> **Main Idea:** *We can convert a numeric vector to a factor*

In this section, we'll investigate the `cut()` function.

The `cut()` function is a technique for converting a set of numeric values into a categorical variable.

We do this by defining a set of numeric ranges, which form the categories.

Then each numeric value is converted to the range in which it occurs.

The is a very common procedure:

- One common form of this is to collapse a person's age into the categories 20-29, 30-39, etc.
- We can categorize households as lower-income, middle-income, or upper-income, depending on their income.
- We can categorize blood pressure, a numeric variable, into two categories, high-blood pressure or normal.

For example, suppose a business has a vector consisting of the ages of its customers:

```r
age.vector <-
    c(
        21.4, 33.8, 27.5, 19.4, 36.9,
        18.3, 24.4, 25.8, 34.6, 26.1
    )
```

We could create the categories “Less than or equal to 20”, “Strictly greater than 20 to less than or equal to 30”, and “Strictly greater than 30” and then assign each element of `age.vector` to its appropriate category, ending up with a table:

Age Category	Number
Age <= 20	1
20 < Age <= 30	2
30 < Age	3

So we’ve generated the categorical variable `Age Category` by choosing a set of ranges and then assigning each continuous variable to its appropriate range.

This process is called “binning”: a “bin” is a container or a box, and we create a finite number of these containers and then put the continuous values into its corresponding bin.

You might wonder how we chose the range intervals – why did we work with three categories “Age <= 20”, “20 < Age <= 30”, and “30 < Age”?

Could we have used something else? And if so, how do we decide on the category ranges?

The answer is that the choice of the range intervals is completely arbitrary, and we could have used another set of ranges.

For instance, we could have used just two categories: “Less than or equal to 25” and “Strictly greater than 25”, in which case our resulting numeric codes would have been:

Age Category	Number
Less than or equal to 25	1
Strictly greater than 25	2

In general, although the range intervals are arbitrary, it’s a good idea to make them sensible numbers so that our results are easy to understand.

Suppose instead that we had chosen to use 2 bins, but now the two categories are “Less than or equal to 23.7” and “Strictly greater than 23.7”.

We’ll still have the same numeric codes as before:

Age Category	Number
Less than or equal to 23.7	1
Strictly greater than 23.7	2

Is that wrong, or mathematically incorrect?

No.

It’s perfectly fine, from a purely mathematical standpoint.

Is it weird?

Yes.

It’s very difficult for people to think about such a category.

In the end, the point of data analysis is to communicate our findings to others, and using unorthodox ranges makes that difficult.

So the choice of the range is really up to the discretion of the data analyst, but the best practice is usually to use a conventional set of ranges.

R provides us with a nice function for performing the binning operation called `cut()`.

This function takes a vector of numeric values and another vector of interval endpoints called `breaks` and generates a factor using those breaks:

```
age.factor <-
    cut(
        x = age.vector,
        breaks = c(0, 20, 30, 40)
    )

age.factor
```

```
##  [1] (20,30] (30,40] (20,30] (0,20]  (30,40] (0,20]  (20,30] (20,30] (30,40]
## [10] (20,30]
## Levels: (0,20] (20,30] (30,40]
```

What happened here?

We started with `age.vector`, and then we specified a set of cut points to define the bins.

Notice that how the levels are denoted: the expression "(0, 20]" means that it consists of values that are strictly greater than 0, and less than or equal to 20.

The R documentation for the `cut()` function refers to this as "closed on the right, open on the left".

If you want the opposite (i.e. closed on the left, open on the right) then you can use the `right = FALSE` option:

```
age.factor.1 <-
    cut(
        age.vector,
        breaks = c(0, 20, 30, 40),
        right = FALSE
    )

age.factor.1
```

```
##  [1] [20,30) [30,40) [20,30) [0,20)  [30,40) [0,20)  [20,30) [20,30) [30,40)
## [10] [20,30)
## Levels: [0,20) [20,30) [30,40)
```

If you set the breaks so that they do not include all the data values, the excluded values will be treated as an `NA`:

```
age.factor.2 <-
    cut(
        age.vector,
        breaks = c(20, 30, 40)
    )

age.factor.2
```

```
##  [1] (20,30] (30,40] (20,30] <NA>    (30,40] <NA>    (20,30] (20,30] (30,40]
## [10] (20,30]
## Levels: (20,30] (30,40]
```

Earlier in the semester, we saw the special values `Inf` and `-Inf`, and I mentioned that occasionally there were legitimate uses for these values.

The `breaks` option in the `cut()` function is one such example, and by using `Inf` and `-Inf` we can insure that all values are included in a bin:

```
age.factor.3 <-
    cut(
        age.vector,
        breaks =
            c(-Inf, 20, 30, Inf)
    )

age.factor.3
```

```
##  [1] (20,30]   (30, Inf] (20,30]   (-Inf,20] (30, Inf] (-Inf,20] (20,30]
##  [8] (20,30]   (30, Inf] (20,30]
## Levels: (-Inf,20] (20,30] (30, Inf]
```

In all of these examples, the `cut()` function creates a factor with levels that clearly and accurately describe the interval ranges.

That's great, and it's often times very nice to have that reminder.

But sometimes you might want a different set of levels for the factor, and you can explicitly specify the factor levels by using the `labels =` option:

```
age.factor.4 <-
    cut(
        x = age.vector,
        breaks =
            c(-Inf, 20, 30, Inf),
        labels =
            c( "Low", "Medium", "High" )
    )

age.factor.4
```

```
##  [1] Medium High   Medium Low    High   Low    Medium Medium High   Medium
## Levels: Low Medium High
```

So that's how to convert a numeric vector to a factor.

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

Exercise 2.5: Creating factors from numeric vectors

Suppose you have a set of final course scores:

```r
final.course.scores <-
    c( 92.1, 85.4, 78.7, 91.3,
       90.0, 93.0, 61.1, 80.0 )
```

Let's use a simplified grading scheme:

- If a student scores 90 or above, the student receives an "A" letter grade.
- If the student scores 80 or above, but less than 90, then the student receives a "B" letter grade.
- If the student scores 70 or above, but less than 90, then the student receives a "C" letter grade.
- If the student scores strictly less than 70, then the student receives an "E" letter grade.

Convert the vector of final course scores to letter grades using the `cut` function. Be careful about how your handle the endpoints.

Solution

Module Review

In this module, we learned how to work with factors.

- In Section 1, we saw how to specify the levels of a factor.
- In Section 2, we learned how to specify the labels of a factor.
- In Section 3, we saw how to re-order the levels.
- In Section 4, we learned how to group levels together.
- In Section 5, we investigated the `cut()` function.

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

- Specify the levels of a factor.
- Specify the labels of a factor.
- Re-order the levels of a factor.
- Group factor levels together.
- Create a factor from a numeric vector.

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

- `levels()`
- `cut()`

All right, that's it for Module 2: Factors.

Now let's move on to Module 3: Tables.

Solutions to the Exercises

Exercise 2.1: Specifying the levels

Suppose we have a very simple vector of character string labels:

```
simple.character.string.vector <-
  c( "blue", "blue", "green", "red", "green", "blue", "red" )
```

Construct a factor that consists of just the values "green" and "blue" by using the `levels` option.

Solution

```
simple.factor <-
    factor(
        simple.character.string.vector,
        levels = c("green", "blue")
    )

simple.factor
```

```
## [1] blue  blue  green <NA>  green blue  <NA>
## Levels: green blue
```

Exercise 2.2: Changing the Levels

Using the vector `simple.character.string.vector`, construct a factor with two levels, which have the labels "Green" and "Blue".

Specify the levels using the `levels` option, and specify the labels for the levels using the `labels` option.

Solution

```
simple.factor <-
    factor(
        simple.character.string.vector,
        levels = c("green", "blue"),
        labels = c( "Green", "Blue" )
    )

simple.factor
```

```
## [1] Blue  Blue  Green <NA>  Green Blue  <NA>
## Levels: Green Blue
```

Exercise 2.3: Re-ordering the levels

Re-order the levels of `simple.character.factor` so that "Blue" is the first level and "Green" is the second level.

Solution

```r
reordered.simple.factor <-
    factor(
        simple.factor,
        levels = c( "Blue", "Green" )
    )

reordered.simple.factor
```

```
## [1] Blue  Blue  Green <NA>  Green Blue  <NA>
## Levels: Blue Green
```

Exercise 2.4: Grouping levels

Suppose we have a sequence of cereal box sales stored in a vector:

```r
cereal.sales.vector <-
    c( "sbz", "sbz", "kym", "hkt",
       "sbz", "kym", "sb", "ky" )

cereal.sales.factor <-
    factor( cereal.sales.vector )
```

We want to represent these sales using our standard 3-letter abbreviation.

Notice that the last 2 entries only use 2 letters for the abbreviation, and R will treat these as distinct levels:

```r
cereal.sales.factor
```

```
## [1] sbz sbz kym hkt sbz kym sb  ky
## Levels: hkt ky kym sb sbz
```

Group the levels “sbz” and “sb” together as “sbz”. Likewise, group the levels “kym” and “ky” together as “kym”.

Solution

```r
levels( cereal.sales.factor ) <-
    c( "hkt", "kym", "kym", "sbz", "sbz" )

cereal.sales.factor
```

```
## [1] sbz sbz kym hkt sbz kym sbz kym
## Levels: hkt kym sbz
```

Exercise 2.5: Creating factors from numeric vectors

Suppose you have a set of final course scores:

```r
final.course.scores <-
    c( 92.1, 85.4, 78.7, 91.3,
       90.0, 93.0, 61.1, 80.0 )
```

Let’s use a simplified grading scheme:

- If a student scores 90 or above, the student receives an “A” letter grade.
- If the student scores 80 or above, but less than 90, then the student receives a “B” letter grade.
- If the student scores 70 or above, but less than 90, then the student receives a “C” letter grade.
- If the student scores strictly less than 70, then the student receives an “E” letter grade.

Convert the vector of final course scores to letter grades using the `cut` function. Be careful about how your handle the endpoints.

Solution

To solve this problem, we need to use the `cut()` function.

First, we’ll construct a vector with the cut points:

```
cut.points.vector <-
    c(0, 70, 80, 90, 100)
```

Now let’s try using the `cut()` function with the `final.course.scores` vector, using the cutpoints in the `cut.points.vector`:

```
cut(
    final.course.scores,
    breaks = cut.points.vector
)
```

```
## [1] (90,100] (80,90]  (70,80]  (90,100] (80,90]  (90,100] (0,70]   (70,80]
## Levels: (0,70] (70,80] (80,90] (90,100]
```

Now let’s put in some labels:

```
cut(
    final.course.scores,
    labels = c( "E", "C", "B", "A" ),
    breaks = cut.points.vector
)
```

```
## [1] A B C A B A E C
## Levels: E C B A
```

But this isn’t quite right – the fifth student scored a 90.0, and therefore should receive an “A” grade, but the code assigns a “B” letter grade to this student. Similarly, the eighth student scored an 80.0, and therefore should receive a “B”, but in fact was assigned a “C”. That’s because these grades are right on the cut points, and the default setting is that the intervals for the `cut()` function are open on the left and closed on the right. That means that the interval for a “B” grade is the range strictly greater than 80 and less than or equal to 90. But that’s not how we specified the grading scheme! Instead, we want the intervals to be closed on the left and open on the right. To do this, we need to set the `right` option to `FALSE`:

```
cut(
    final.course.scores,
    labels = c( "E", "C", "B", "A" ),
    breaks = cut.points.vector,
    right = FALSE
)
```

```
## [1] A B C A A A E B
## Levels: E C B A
```

Now it works properly!