MTH 2520 R Notes 7

7 Factors and Tables

7.1 Creating and Viewing Factors and Their Levels

- *Factors*, like `"character"` vectors, are used to store categorical (qualitative) data. They differ from `"character"` vectors in the way R stores them internally, and they contain a bit of extra information called *levels*.
- To create and look at factors, the following are useful:

```
factor()          # Create a factor from a character vector
length()          # Returns the number of elements in a factor
levels()          # Examine the levels of the factor
nlevels()         # Returns the number of levels of the factor
is.factor()       # Indicates whether or not an object is a factor
str()             # Describes the structure of a factor
```

- `factor()` takes a `"character"` vector and converts it to a factor:

```
> char.vec <- c("ctrl", "trt1", "trt2", "ctrl", "trt1", "trt2", "ctrl", "trt1", "trt2")

> my.fac <- factor(char.vec)

> is.factor(my.fac)

[1] TRUE
```

- When R prints out a factor, it also indicates its ***levels***, or unique values:

```
> my.fac

[1] ctrl trt1 trt2 ctrl trt1 trt2 ctrl trt1 trt2
Levels: ctrl trt1 trt2
```

- To convert a factor to a `"character"` vector, we use:

```
as.character()      # Convert a factor to a character vector
```

- For example:

```
> char.vec <- as.character(my.fac)

> is.vector(char.vec)

[1] TRUE
```

- To see how many elements a factor has, use `length()`. To look at the *levels* of a factor, use `levels()`. To simply see *how many* levels there are, use `nlevels()`.

```
> length(my.fac)

[1] 9

> levels(my.fac)

[1] "ctrl" "trt1" "trt2"

> nlevels(my.fac)

[1] 3
```

Section 7.1 Exercises

Exercise 1

a) Write a command that uses `factor()` to create the following factor, `x.factor`:

```
> x.factor

 [1] a a b b c c d d
Levels: a b c d
```

b) After creating the factor `x.factor`, guess what the result of the following command will be, then check your answer:

```
> length(x.factor)
```

c) Guess what the result of the following command will be, then check your answer:

```
> levels(x.factor)
```

d) Guess what the result of the following command will be, then check your answer:

```
> nlevels(x.factor)
```

Exercise 2 Guess what the result of the following commands will be, then check your answer:

```
> x.fac <- factor(c("a", "a", "a", "b", "b", "c"))
> levels(x.fac)
```

7.2 The Difference Between Factors and "character" Vectors

- Unlike "character" vectors, R stores factors internally as *numeric vectors*, with integer values representing the levels of the factor.

  To see, recall that typeof() indicates the *type* of an object, and for the factor my.fac and "character" vector char.vec from above, returns:

```
> typeof(my.fac)

[1] "integer"

> typeof(char.vec)

[1] "character"
```

- The integer values used internally to represent the levels of a factor can be viewed using:

```
unclass()          # Remove the class attribute of an object.  Can be
                   # used to view the integer values used internally
                   # to represent the levels of a factor.
```

- For example, here's my.fac again:

```
> my.fac

[1] ctrl trt1 trt2 ctrl trt1 trt2 ctrl trt1 trt2
Levels: ctrl trt1 trt2
```

  The integers used to represent the levels of my.fac can be viewed by typing:

```
> unclass(my.fac)

[1] 1 2 3 1 2 3 1 2 3
attr(,"levels")
[1] "ctrl" "trt1" "trt2"
```

The numbering is as follows. An observation made at the jth level of the factor (i.e. the level in the jth position as returned by `levels()`) is represented by the value j. Above, because the levels of `my.fac` (as returned by `levels()`) are `"ctrl"`, `"trt1"`, and `"trt2"`, observations made under the `"ctrl"` condition are represented by the value 1, those made under `"trt1"` condition by the value 2, and those made under the `"trt2"` condition by the value 3.

Section 7.2 Exercises

Exercise 3 Here's a factor:

```
> y.fac <- factor(c("b", "b", "c", "a", "c", "a"))
> y.fac
```

```
[1] b b c a c a
Levels: a b c
```

Guess what the integer representation of `y.fac` is, then check your answer by typing:

```
> unclass(y.fac)
```

Exercise 4 Here's a factor:

```
> y.fac <- factor(c("e", "f", "e", "g", "g", "f"))
> y.fac
```

```
[1] e f e g g f
Levels: e f g
```

Guess what the integer representation of `y.fac` is, then check your answer by typing:

```
> unclass(y.fac)
```

Exercise 5 Here's a factor:

```
> y.fac <- factor(c("high", "med", "low", "high", "low", "med"))
> y.fac
```

```
[1] high med  low  high low  med
Levels: high low med
```

Guess what the integer representation of `y.fac` is, then check your answer by typing:

```
> unclass(y.fac)
```

Exercise 6 Write out the factor `y.fac` based on the information given below:

```
> unclass(y.fac)


[1] 2 3 2 1 1 3
attr(,"levels")
[1] "a" "b" "c"
```

Exercise 7 Write out the factor `y.fac` based on the information given below:

```
> unclass(y.fac)


[1] 2 1 3 3 2 1
attr(,"levels")
[1] "high" "low"  "med"
```

Exercise 8 Factors can be annoying because R can convert a factor to a numeric vector without telling you. For example, here's a factor:

```
> y.fac <- factor(c("a", "a", "b", "b", "c", "c"))
```

What is the result of

```
> c(y.fac, 6)
```

Try it, and explain the result.

7.3 Changing the Levels of a Factor

7.3.1 Reordering the Levels

- The order of the levels of the factor, as returned by `levels()`, determines the order in which they appear in tables (see below) and plots (later).
- To set the order when creating the factor, use the (optional) `levels` argument in the call to `factor()`:

  ```
  > my.fac <- factor(char.vec, levels = c("trt1", "trt2", "ctrl"))
  > levels(my.fac)


  [1] "trt1" "trt2" "ctrl"
  ```

  (The default order was `"ctrl"`, `"trt1"`, `"trt2"`).
- To change the order of levels in an existing factor, we again use the `levels` argument in `factor()`:

  ```
  > my.fac <- factor(my.fac, levels = levels(my.fac)[c(3, 1, 2)])
  > levels(my.fac)
  ```

```
[1] "ctrl" "trt1" "trt2"
```

7.3.2 Anticipating Additional Levels

- If we try to insert a value in a factor and that value isn't already one of the factor levels, R inserts an `NA` and prints a warning message:

```
> levels(my.fac)

[1] "ctrl" "trt1" "trt2"

> my.fac[10] <- "trt3"

Warning message:
In `[<-.factor`(`*tmp*`, 10, value = "trt3") :
  invalid factor level, NA generated
```

- Instead, we need to create the level first, in anticipation of the value being inserted, using the `levels` argument to `factor()`:

```
> my.fac <- factor(char.vec, levels = c("ctrl", "trt1", "trt2", "trt3"))
> levels(my.fac)

[1] "ctrl" "trt1" "trt2" "trt3"

> my.fac[10] <- "trt3"
```

7.3.3 Removing Unwanted Factor Levels

- Factors can be annoying. When you take a subset of a factor, for example using square brackets `[ ]`, R keeps the original levels, even if they're not they're represented in the subset:

```
> my.fac <- factor(char.vec)
> my.fac

[1] ctrl trt1 trt2 ctrl trt1 trt2 ctrl trt1 trt2
Levels: ctrl trt1 trt2

> my.new.fac <- my.fac[1:2]
> my.new.fac

[1] ctrl trt1
Levels: ctrl trt1 trt2
```

Note that `trt2` is still included among the levels of `my.new.fac`, even though `"trt2"` isn't one of the values included in the factor itself.

This can affect the appearance of tables (see below) and plots (later) that are based on the subset of the factor.

- One remedy is to re-define the factor using `factor()`, as below:

```
> my.new.fac <- factor(my.fac[1:2])
> my.new.fac

[1] ctrl trt1
Levels: ctrl trt1
```

Section 7.3 Exercises

Exercise 9 Here's a factor:

```
> w.char.vec <- c("a", "a", "b", "b", "c", "c")
> x.fac <- factor(w.char.vec)
> x.fac

[1] a a b b c c
Levels: a b c
```

a) Guess what the result of the following command will be, then check your answer:

```
> x.factor[7] <- "d"
```

b) How could we assign the value `"d"` as the 7th element of `x.factor`? Write R commands to do so.

Exercise 10 When you take a subset of a factor, for example using square brackets `[ ]`, R keeps the original levels even if they aren't represented in the subset.

a) Guess what levels will be returned by `levels()` below, then check your answer:

```
> x.fac <- factor(c("a", "a", "a", "b", "b", "c"))
> new.x.fac <- x.fac[1:5]
> levels(new.x.fac)
```

b) This can be annoying when creating tables and plots from factors. The `table()` function will count how many times each level appears among the elements of a factor:

```
> new.x.fac

[1] a a a b b
Levels: a b c
```

```
> table(new.x.fac)

new.x.fac
a b c
3 2 0
```

Notice that the table includes a count for level `"c"` even though `"c"` doesn't appear among the elements of `new.x.fac`.

How would you remove the level `"c"` from `new.x.fac` so it doesn't show up in the table?

Exercise 11 The order of the levels of the factor, as returned by `levels()`, determines the order in which they appear in tables (see the last exercise) and plots.

Here's a factor containing peoples' political party affiliations:

```
> party <- factor(c("R", "D", "I", "R", "D", "R", "I", "D", "D"))
> levels(party)

[1] "D" "I" "R"
```

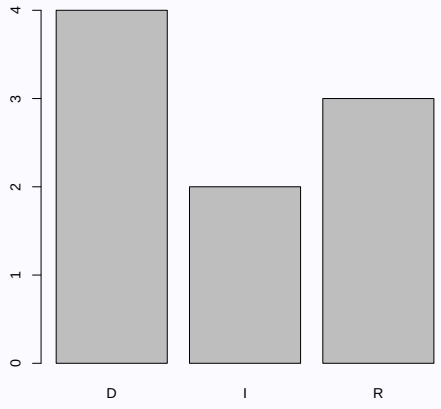
The `table()` function will make a table from the counts in a factor:

```
> party.tab <- table(party)
> party.tab

party
D I R
4 2 3
```

The `barplot()` function will make a bar plot from the counts in a table:

```
> barplot(party.tab)
```



The ordering of the bars in the plot is determined by the ordering of the levels in the `party` factor, as returned by `levels()`.

How would you reorder the levels in the `party` factor so that they appear in the order `"D"`, `"R"`, `"I"` in the plot?

7.4 The `split()` Function

- The function `split()` will split a vector (or data frame) into groups defined by a factor (or `"character"` vector):

```
split()        # Split a vector or data frame into to groups defined
               # by a factor (or "character" vector)
```

- `split()` takes arguments `x`, a vector (or data frame), and `f`, a factor (or `"character"` vector) defining groups. It returns a list whose elements are vectors (or data frames) corresponding to the groups.
- For example, here's a vector of responses from an experiment, and a factor indicating the treatment groups:

```
> Response <- c(23, 11, 14, 16, 19, 26, 24, 29, 31, 28, 34, 25)
> Group <- factor(c(rep("ctrl", 4), rep("trt1", 4), rep("trt2", 4)))

> Response

 [1] 23 11 14 16 19 26 24 29 31 28 34 25

> Group

 [1] ctrl ctrl ctrl ctrl trt1 trt1 trt1 trt1 trt2 trt2 trt2 trt2
Levels: ctrl trt1 trt2
```

To split the `Response`s according to the treatment `Group`s, we type:

```
> split(x = Response, f = Group)

$ctrl
[1] 23 11 14 16

$trt1
[1] 19 26 24 29

$trt2
[1] 31 28 34 25
```

Note that `split()` returns a *list* of vectors, each of which corresponds to a `Group`.

- Splitting a data frame is done in a similar manner. For example, here's a data frame containing the experiment data, along with some additional information about the subjects:

```
> experiment.data
```

```
   Group Gender Age Response
1   ctrl      m  33       23
2   ctrl      m  45       11
3   ctrl      f  30       14
4   ctrl      f  24       16
5   trt1      m  22       19
6   trt1      f  31       26
7   trt1      f  39       24
8   trt1      m  40       29
9   trt2      f  29       31
10  trt2      m  19       28
11  trt2      f  27       34
12  trt2      m  25       25
```

To split the `experiment.data` according to the `Group`, we type:

```
> split(x = experiment.data, f = experiment.data$Group)
```

```
$ctrl
  Group Gender Age Response
1  ctrl      m  33       23
2  ctrl      m  45       11
3  ctrl      f  30       14
4  ctrl      f  24       16

$trt1
  Group Gender Age Response
5  trt1      m  22       19
6  trt1      f  31       26
7  trt1      f  39       24
8  trt1      m  40       29

$trt2
   Group Gender Age Response
9   trt2      f  29       31
10  trt2      m  19       28
11  trt2      f  27       34
12  trt2      m  25       25
```

Notice that `split()` returns a *list* of data frames, each of which corresponds to a `Group`.

Section 7.4 Exercises

Exercise 12 Here's a vector and a factor:

```
> w.vec <- c(2, 1, 5, 2, 3, 6, 7, 9, 4, 3, 6, 8)
> x.fac <- factor(c("c", "b", "a", "a", "c", "b", "a", "b", "c",
                    "a", "c", "b"))
```

a) Write a command involving `split()` that splits `w.vec` into groups defined by `x.fac`.

b) What type of object (vector, matrix, array, list, or data frame) does `split()` return?

c) Use the output from `split()`, along with `mean()` and `sd()`, to find the mean and standard deviation for each group.

Exercise 13 Here's a data frame containing data from an experiment involving a treatment group and a control group:

```
> x <- data.frame(Group = c("Trt", "Trt", "Trt", "Trt", "Ctrl", "Ctrl",
                            "Ctrl", "Ctrl"),
                  Y = c(22, 45, 32, 30, 44, 24, 56, 59))
> x
```

```
  Group  Y
1   Trt 22
2   Trt 45
3   Trt 32
4   Trt 30
5  Ctrl 44
6  Ctrl 24
7  Ctrl 56
8  Ctrl 59
```

a) Write a command involving `split()` that splits `x` into two separate data frames, one corresponding to the treatment group and the other to the controls.

b) What type of object (vector, matrix, array, list, or data frame) does `split()` return?

Exercise 14 Recall that the built-in R data frame `warpbreaks` contains data from a study of the strength of yarn used in weaving. There are three variables in the data set:

`breaks`	The number of breaks per loom, where a loom corresponds to a fixed length of yarn.
`wool`	The type of wool (A or B)
`tension`	The level of tension (L, M, H)

To look at the data, type:

```
> warpbreaks
```

and to read about it, look at its help file:

```
> ? warpbreaks
```

We can split a data frame according to *two* grouping variables by passing them as *list* via the argument `f` in `split()`. Guess what the following command will return, then check your answer:

```
> split(warpbreaks, f = list(warpbreaks$wool, warpbreaks$tension))
```

Exercise 15 Recall from Exercise 14 that we can split a data frame according to *two* grouping variables by passing them as *list* via the argument `f` in `split()`. Here's a data frame containing data from an experiment involving a treatment group and a control group:

```
> x <- data.frame(Group = c("Trt", "Trt", "Trt", "Trt", "Ctrl", "Ctrl",
                            "Ctrl", "Ctrl"),
                  Gender = c("M", "M", "F", "F", "M", "M", "F", "F"),
                  Y = c(22, 45, 32, 30, 44, 24, 56, 59))
> x
```

```
  Group Gender  Y
1   Trt      M 22
2   Trt      M 45
3   Trt      F 32
4   Trt      F 30
5  Ctrl      M 44
6  Ctrl      M 24
7  Ctrl      F 56
8  Ctrl      F 59
```

Write a command involving `split()` that splits the data set into four smaller data frames defined by the treatment received (treatment vs control) and gender of the subject (male vs female). You should end up with this:

```
$Ctrl.F
  Group Gender  Y
7  Ctrl      F 56
8  Ctrl      F 59

$Trt.F
  Group Gender  Y
3   Trt      F 32
4   Trt      F 30

$Ctrl.M
  Group Gender  Y
5  Ctrl      M 44
```

```
6  Ctrl      M 24

$Trt.M
  Group Gender  Y
1   Trt      M 22
2   Trt      M 45
```

7.5 The `tapply()` Function

- The function `tapply()` is used to apply a function to subsets of a vector corresponding to groups defined by a factor (or `"character"` vector):

```
tapply()     # Apply a function to subsets of a vector that correspond
             # to groups defined by a factor (or "character" vector)
```

- `tapply()` takes arguments `X`, a vector, `INDEX`, a factor defining the groups, and `FUN`, a function, and applies the function separately for each group.
- For example, here's a vector of responses from an experiment, and a factor indicating the treatment groups:

```
> Response <- c(23, 11, 14, 16, 19, 26, 24, 29, 31, 28, 34, 25)
> Group <- factor(c(rep("ctrl", 4), rep("trt1", 4), rep("trt2", 4)))

> Response

 [1] 23 11 14 16 19 26 24 29 31 28 34 25

> Group

 [1] ctrl ctrl ctrl ctrl trt1 trt1 trt1 trt1 trt2 trt2 trt2 trt2
Levels: ctrl trt1 trt2
```

To compute the mean `Response` for each treatment `Group`, type:

```
> tapply(X = Response, INDEX = Group, FUN = mean)

ctrl trt1 trt2
16.0 24.5 29.5
```

Although it may not be apparent, `tapply()` returns its results in an *array* (one-dimensional above).

- If the data are in a data frame, we need to use the dollar sign `$` (or double square brackets `[[ ]]`) when referring to the vector and factor. For example, here's the experiment data frame:

```
> experiment.data
```

```
   Group Gender Age Response
1   ctrl      m  33       23
2   ctrl      m  45       11
3   ctrl      f  30       14
4   ctrl      f  24       16
5   trt1      m  22       19
6   trt1      f  31       26
7   trt1      f  39       24
8   trt1      m  40       29
9   trt2      f  29       31
10  trt2      m  19       28
11  trt2      f  27       34
12  trt2      m  25       25
```

To compute the treatment group means, we type:

```
> tapply(X = experiment.data$Response, INDEX = experiment.data$Group, FUN = mean)
```

```
ctrl trt1 trt2
16.0 24.5 29.5
```

Section 7.5 Exercises

Exercise 16 Here's a vector and a factor:

```
> x.vec <- c(2, 1, 5, 2, 3, 6, 7, 9, 4, 3, 6, 8)
> x.char.vec <- c("c", "b", "a", "a", "c", "b", "a", "b", "c",
                  "a", "c", "b")
> x.fac <- factor(x.char.vec)
```

After creating `x.vec` and `x.fac`, write a command involving `tapply()` that computes the mean of `x.vec` separately for each group defined by `x.fac`.

Exercise 17 Here's a data frame containing data from an experiment involving a treatment group and a control group:

```
> x.df <- data.frame(Group = c("Trt", "Trt", "Trt", "Trt", "Ctrl", "Ctrl",
                               "Ctrl", "Ctrl"),
                     Y = c(22, 45, 32, 30, 44, 24, 56, 59))
> x.df
```

```
  Group  Y
1   Trt 22
2   Trt 45
3   Trt 32
```

```
4   Trt 30
5  Ctrl 44
6  Ctrl 24
7  Ctrl 56
8  Ctrl 59
```

After creating the data frame `x.df`, Write a command involving `tapply()` that computes the mean of `Y` separately for each of the treatment `Groups`.

Exercise 18 Consider again the built-in `warpbreaks` data frame described in Exercisese 14 and 15.

We can use `tapply()` to apply a function to values in a vector separately for groups defined by *two* grouping variables by passing them as *list* via the argument `INDEX` in `tapply()`.

What do you think the following command will do? Try it.

```
> tapply(X = warpbreaks$breaks,
         INDEX = list(warpbreaks$wool, warpbreaks$tension),
         FUN = mean)
```

Exercise 19 Recall from Exercise 18 that we can apply a function to values in a vector separately for groups defined by *two* grouping by passing them as *list* via `INDEX` in `tapply()`.

Here's a data frame containing data from an experiment involving a treatment group and a control group:

```
> x <- data.frame(Group = c("Trt", "Trt", "Trt", "Trt", "Ctrl", "Ctrl",
                            "Ctrl", "Ctrl"),
                  Gender = c("M", "M", "F", "F", "M", "M", "F", "F"),
                  Y = c(22, 45, 32, 30, 44, 24, 56, 59))
> x

  Group Gender  Y
1   Trt      M 22
2   Trt      M 45
3   Trt      F 32
4   Trt      F 30
5  Ctrl      M 44
6  Ctrl      M 24
7  Ctrl      F 56
8  Ctrl      F 59
```

Write a command involving `tapply()` that computes the mean of `Y` separately for groups defined by the treatment received (treatment vs control) and the gender of the subject (male vs female). You should end up with this:

```
        F    M
Ctrl 57.5 34.0
Trt  31.0 33.5
```

7.6 Creating Tables

- To create a table summarizing categorical data (factor or `"character"` vector), we use:

```
table()          # Create a table of counts from a factor or
                 # "character" vector
prop.table()     # Create a table of proportions from a table
                 # counts
```

To check whether an object is a table, use:

```
is.table()       # Returns TRUE or FALSE indicating whether an object
                 # is a table
```

To turn the counts (or proportions) in a table into a bar graph, we use:

```
barplot()        # Create a bar graph from a table of counts or
                 # proportions
```

- `table()` counts the number of observations at each level of the factor (or unique value in the `"character"` vector), and returns the results in a table.
- For example, here's a `"character"` vector containing responses (Yes/No/NotSure) to a survey question:

```
> survey.responses <- c("Yes", "No", "NotSure", "No", "NotSure", "NotSure",
+     "No", "Yes", "No", "Yes", "NotSure", "No", "Yes", "Yes", "Yes",
+     "NotSure", "Yes", "No", "Yes")
```

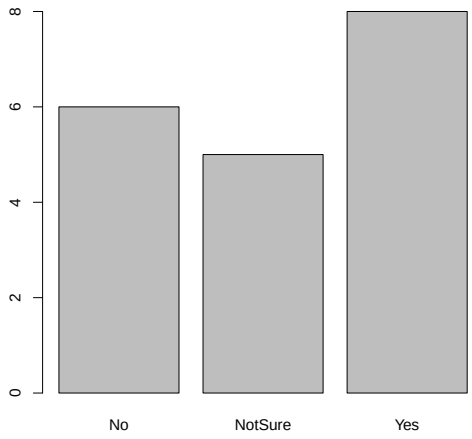
We can tabulate the responses by typing:

```
> survey.tab <- table(survey.responses)
> survey.tab

survey.responses
     No NotSure     Yes
      6       5       8
```

Here's the bar graph:

```
> barplot(survey.tab)
```



- `prop.table()` takes the table of counts returned by `table()`, and converts the counts to *proportions*:

```
> prop.table(survey.tab)

survey.responses
       No   NotSure       Yes
0.3157895 0.2631579 0.4210526
```

- We can create a *two-way* table from *two* factors (or `"character"` vectors). For example, here's a data frame in which individuals are cross-classified according to `AgeGroup` and political `Affiliation`:

```
> political.data

   AgeGroup Affiliation
1     Young    Democrat
2     Young  Republican
3       Old  Republican
4       Old  Republican
5     Young    Democrat
6     Young  Republican
7       Old    Democrat
8       Old  Republican
9       Old  Republican
10    Young    Democrat
```

To create a two-way table, we type:

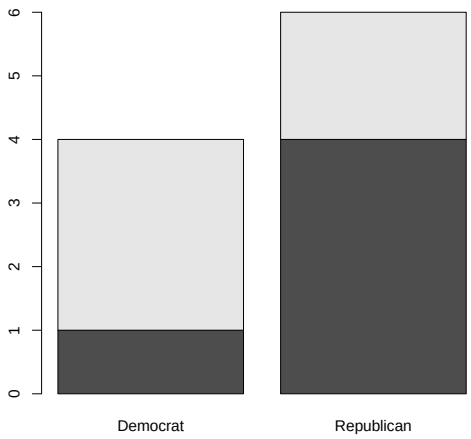
```
> pol.tab <- table(political.data$AgeGroup, political.data$Affiliation)
> pol.tab
```

```
      Democrat Republican
Old          1          4
Young        3          2
```

(Above, we could also have just passed the entire data from to `table()`, i.e. we could've used `table(political.data)`.)

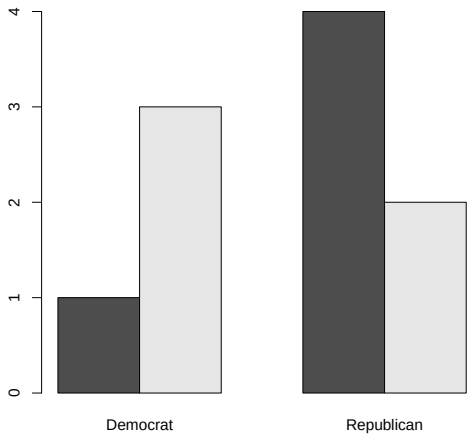
In this case, the bar graph looks like this:

```
> barplot(pol.tab)
```



By setting an optional argument `beside` to `TRUE` in `barplot()`, we get a graph with bars beside each other:

```
> barplot(pol.tab, beside = TRUE)
```



- It turns out that *tables* are also *arrays*:

```
> is.table(pol.tab)

[1] TRUE

> is.array(pol.tab)

[1] TRUE
```

Section 7.6 Exercises

Exercise 20 The built-in data set `state.region` is a factor indicating the region (Northeast, South, etc.) for each of the 50 states in the U.S.

a) Use `table()` to count the number of states in each region.

b) Use `prop.table()` to determine the *proportion* of states that belong to each region.

c) Write a command involving `barplot()` that creates a bar graph from the counts in the table of part *a*.

Exercise 21 Another of R's built-in data sets (in addition to `state.region`) is `state.division`, a factor indicating the division (New England, Middle Atlantic, etc.) for each of the 50 states.

a) Use `table()` create a two-way table of counts giving the number of states cross-classified according to region and division.

b) Use `prop.table()` to determine the *proportion* of states in each region by division cross-classification.

c) Write a command involving `barplot()` that creates a bar graph from the counts in in the table of part *a*.

7.7 The `cut()` Function

- The function `cut()` is used to create a factor by grouping values from a numeric vector into "bins" (intervals):

```
cut()     # Create a factor from a numeric vector by grouping values
          # into bins
```

- `cut()` takes arguments `x`, a numeric vector, and `breaks`, a vector of cut points for the bins (intervals), and returns a factor indicating which bin each element of `x` belongs to.
- For example, here's a vector of values in the range 0 to 100:

```
> data.vec

 [1] 34 25 63 55 33 40 58 44 79 76 33 17 49 76 48
```

To group the elements of `data.vec` into bins (intervals) $(0, 25]$, $(25, 50]$, $(50, 75]$, and $(75, 100]$, we type

```
> cut(x = data.vec, breaks = seq(0, 100, 25))

 [1] (25,50]  (0,25]   (50,75]  (50,75]  (25,50]  (25,50]  (50,75]  (25,50]  (75,100] (75,10
[14] (75,100] (25,50]
Levels: (0,25] (25,50] (50,75] (75,100]
```

Note that the result is a factor.

- An optional argument, `labels`, a `"character"` vector, can be used to make nice labels for the bins:

```
> cut(x = data.vec, breaks = seq(0, 100, 25),
+     labels = c("Low", "MedLow", "MedHigh", "High"))

 [1] MedLow  Low     MedHigh MedHigh MedLow  MedLow  MedHigh MedLow  High    High    MedLow
Levels: Low MedLow MedHigh High
```

Section 7.7 Exercises

Exercise 22 Here's a vector of people's ages:

```
> age <- c(11, 45, 22, 19, 27, 66, 72, 44, 13, 17, 55, 69, 21, 31)
```

a) Write a command involving `cut()` that uses the `age` vector to produce a factor, call it `age.fac`, indicating which of the age bins $(0, 25]$, $(25, 50]$, and $(50, 75]$ each person belongs to.

b) Pass the `age.fac` factor of part *a* to `table()` to summarize the `age`s in a table.

c) Repeat part *a*, but use the argument `labels` in `cut()` to specify the labels `"Young"`, `"Mid"`, and `"Old"` for the three age bins.