# R Programming
# Statistical Data

## Categorical Data

A good deal of statistical data is of a form which indicates which one of several possible categories that an observation falls into.

## Examples

- Eye colour: *brown*, *hazel*, *green*, *blue*.
- Location: *North Island*, *South Island*, *other*.
- Pain level: *low*, *medium*, *high*.

R provides a facility for creating this kind of data through the functions `factor` and `ordered`.

## Factors

The function `factor` creates data objects which represent variables containing *unordered* categorical data. It takes a character or numeric vector as an argument and returns a factor.

```
> eyes = c("hazel", "blue", "brown",
           "green", "blue", "brown")
> eyecol = factor(eyes)

> eyes
[1] "hazel" "blue"  "brown" "green" "blue"
[6] "brown"

> eyecol
[1] hazel blue  brown green blue  brown
Levels: blue brown green hazel
```

## Factor Levels

The set of all possible categories for a set of categorical variable is called the levels of the corresponding factor.

- By default, R takes the levels to be the set of values occurring in the input data vector, *sorted into ascending order* (either numerically or alphabetically).

- When a factor is printed, the levels of the factor are displayed after the variable.

```
> eyecol
[1] hazel blue  brown green blue  brown
Levels: blue brown green hazel
```

## Specifying Factor Levels

The default set of factor levels, and the order they appear in can be specified with a second argument to `factor`.

```
> eyecol = factor(eyes,
                  levels = c("blue", "green",
                             "hazel", "brown"))
> eyecol
[1] hazel blue  brown green blue  brown
Levels: blue green hazel brown
```

The levels of a factor can be obtained with the `levels` function.

```
> levels(eyecol)
[1] "blue"  "green" "hazel" "brown"
```

## Ordered Factors

Sometimes there is a natural order to a factor's levels. In this case factors are described as *ordered factors*.

- Ordered factors are created with the R function `ordered`.
- It is important to specify the levels when creating an ordered factor to ensure that they are in the correct relationship to each other.

```
> pain = ordered(c("low", "medium", "medium",
                   "high", "medium", "low"),
                 levels = c("low", "medium",
                            "high"))
```

## Producing Factors from Numeric Vectors

- The function `cut` can be used to produce factors from numeric vectors.
- This is done by partitioning the range of the numeric vector into bins and recording which bin the observations fall in.
- The bins are specified with a sequence of cut points.

```
> x
 [1] 4.8 8.6 0.6 4.2 2.0 2.2 2.6 3.5 7.0 9.9

> cut(x, seq(0, 10, by = 2))
 [1] (4,6]  (8,10] (0,2]  (4,6]  (0,2]
 [6] (2,4]  (2,4]  (2,4]  (6,8]  (8,10]
5 Levels: (0,2] (2,4] (4,6] ... (8,10]
```

## Printing Factors

For the most part, factors and ordered factors can be used interchangeably. When ordered factors are printed, the levels are printed after the actual values and the ordering of the levels is indicated.

```
> eyecol
[1] hazel blue  brown green blue  brown
Levels: blue green hazel brown

> pain
[1] low    medium medium high   medium low
Levels: low < medium < high
```

## Factors and Vectors

Factors look very much like vectors, and for many purposes they can be treated as such. In fact, they are special kinds of objects which are created from vectors in a similar way to which matrices are created from vectors.

To tell whether a value is a factor, use the functions, `is.factor` and `is.ordered`.

```
> is.factor(eyecol)
[1] TRUE
> is.factor(pain)
[1] TRUE
> is.ordered(eyecol)
[1] FALSE
> is.ordered(pain)
[1] TRUE
```

## Operations on Factors

About the only operation which makes sense with an unordered vector is to compare the values with a particular value using `==` or `!=`.

For ordered factors, comparisons using `<`, `<=`, `>` and `>=` also make sense.

```
> eyecol == "blue"
[1] FALSE  TRUE FALSE FALSE  TRUE FALSE

> eyecol < "blue"
[1] NA NA NA NA NA NA
Warning message:

In Ops.factor(eyecol, "blue") :
    < not meaningful for factors
```

## Subsetting and Factors

Because factors behave as though they were vectors (and internally they are vectors), the same kinds of subsetting operations apply to them. For example, if we have a vector `hgt` which contains the heights of class members and a factor called `sex` which contains the gender of class members then the following expressions make sense

```
hgt[1:10]              # first 10 heights
sex[1:10]              # first 10 genders
hgt[sex == "male"]     # heights of males
hgt[sex == "female"]   # heights of females
sex[hgt > 180]         # genders of tall people
```

## Tabulation

One of the few things that can be done with factors is to count the number of times each level occurs. This can be done with the function `table`.

```
> table(eyecol)
eyecol
 blue green hazel brown
    2     1     1     2

> table(pain)
pain
   low medium   high
     2      3      1
```

## Cross-Tabulation

It is also possible to use `table` to count the number of times each combination of the levels of two (or more) factors occurs.

```
> table(eyecol, pain)
       pain
eyecol   low medium high
  blue     0      2    0
  green    0      0    1
  hazel    1      0    0
  brown    1      1    0
```

The resulting matrix (or more general array) is called a *contingency table*.

## Data Binning

- We have seen that data can be turned into an ordered factor using the `cut` function.
- Tabulation can then be used to determine how many observations fall in each cell.

```
> x
 [1] 6.9 5.2 4.4 6.5 7.7 8.0 0.6 7.1 5.2 4.9

> tabulate(cut(x, seq(0, 10, by = 2)))
[1] 1 0 4 5
```

## Obtaining Summaries over Factor Levels

Factors provide a way of defining subgroups in a data set. It is useful to be able to obtain summaries for these subgroups. The function `tapply` can be used to do this. The function call

```
tapply(variable, factor, summary)
```

returns a vector containing the specified summaries for the given vector, broken down into the subgroups defined by specified factor.

For the class height example, the expression

```
tapply(hgt, sex, mean)
```

will return a vector containing two elements; the average heights for males and females in the class. The values are named by the factor levels.

## More Complex Summaries

The function `tapply` can also obtain summaries broken down by several factors. An expression of the form

```
tapply(variable, list(fac1, fac2, ...),
       summary)
```

will produce an array, giving the summary broken down in the subgroups specified by the combinations of the given factor levels.

## Data Frames

Data frames provide a way of grouping a number of related variables into a single data object. The function `data.frame` takes a number of vectors and/or factors and returns a single object containing all the variables.

```
df = data.frame(var1, var2, ...)
```

Each of the *var1*, *var2*, . . . is either an expression specifying a vector or factor, or a named expression of the form

```
name = var
```

where *name* provides a name for the given variable in the data frame.

## An Example

A simple gender/height data set.

```
> sex = factor(rep(c("female", "male"), each = 4))
> hgt = c(165, 176, 171, 177, 176, 193, 180, 193)
> classinfo = data.frame(sex, hgt)
> classinfo
     sex hgt
1 female 165
2 female 176
3 female 171
4 female 177
5   male 176
6   male 193
7   male 180
8   male 193
```

## Subsetting

Subsets can be extracted from data frames in the same way as from matrices.

```
> classinfo[c(1,3,5,7), ]
     sex hgt
1 female 165
3 female 171
5   male 176
7   male 180

> classinfo[,1]
[1] female female female female male   male
[7] male   male
Levels: female male
```

[ Note that the optional argument `drop=FALSE` ensures the result is a data frame. ]

## Extracting Variables from Data Frames

The underlying representation of data frames is as a named list of vectors and factors. This representation can be used to extract elements by name.

```
> classinfo$hgt
[1] 165 176 171 177 176 193 180 193

> classinfo$sex
[1] female female female female male   male
[7] male   male
Levels: female male

> tapply(classinfo$hgt, classinfo$sex, mean)
female   male
172.25 185.50
```

## Expressions Involving Data Frame Variables

Names like `classinfo$sex` and `classinfo$hgt` can be tiresome to type, and there is a special way of specifying expressions involving variables from data frames.

```
> with(classinfo, tapply(hgt, sex, mean))
female   male
172.25 185.50
```

The first argument to `with` is a data frame. The second is an expression involving the variables from the data frame.

The second argument can be a compound expression grouped using `{` and `}`. (But remember that only the last expression in the compound will be returned as the value of the `with`.)

## Adding Derived Variables to Data Frames

The function `transform` can be used to produce new variables from those already present in a data frame and to combine the old and new variables into a new data frame.

```
> nclass = transform(classinfo, hgt2 = hgt^2)
> nclass
     sex hgt  hgt2
1 female 165 27225
2 female 176 30976
3 female 171 29241
4 female 177 31329
5   male 176 30976
6   male 193 37249
7   male 180 32400
8   male 193 37249
```

## Alternative Subsetting Facilities

Treating data frames as matrices is a little unnatural. The R function `subset` provides an alternative way of extracting subsets.

```
> subset(classinfo, hgt > 190 & sex == "male")
   sex hgt
6 male 193
8 male 193

> subset(classinfo, c(FALSE, TRUE))
     sex hgt
2 female 176
4 female 177
6   male 193
8   male 193
```

## Selecting Variables

There is also a `select` argument to `subset` which can be used to select variables from a data frame. Here is an example selecting cases where `hgt > 170` for the variables `hgt2` and `sex` in the data frame `nclass`.

```
> subset(nclass, hgt > 170,
                 select = c(hgt2, sex))
   hgt2    sex
2 30976 female
3 29241 female
4 31329 female
5 30976   male
6 37249   male
7 32400   male
8 37249   male
```

## The Select Argument

The select argument works as follows. The variable names are first replaced by their column indices and then the expression is evaluated. This means that selections like:

```
c(sex, age:weight, 20:30)
```

will work. The ability to work with variable names rather than column indices can be helpful.

Always be careful, however, to check that you are getting what you think you are getting.

The function `names` will get the (vector of) names of the variables in a data frame. This can be helpful.

## Reading Data

- The standard way of storing statistical data is to store them in a rectangular form with rows corresponding to observations and columns corresponding to variables.
- Spreadsheets are often used to store and manipulate data in this way.
- The function `read.table` can be used to read data which has been stored in this way.
- The first argument to `read.table` identifies the file to be read.

## File Names and Path Names

- A simple file name (i.e. one not containing `/` in Linux or `\` in Windows) identifies a file in directory where R was invoked.
- Files in other places can be specified with a relative or absolute path name.
- In Linux, path names have the form *dir*`/`. . . `/`*dir*`/`*file* where *dir* is a directory name and *file* is a file name.
  - A leading `/` indicates the top level of the file system.
  - A leading `~` indicates the user's home directory.
  - The character `.` represents the current directory.
  - The character `..` represents the directory one level up from the current one.

## File Names and Path Names

- The specification `../data/foo.txt` specifies the file called `foo.txt` which can be located by going up one level and then down into a directory called `data`.
- Windows is somewhat brain-damaged when it comes to file locations and the best thing to do is to specify `file.choose()` as the file name and then to select the appropriate file using the resulting dialog box.
- Alternatively there is a way of setting a current directory under the file menu.

## Customised Use of `read.table`

There are many optional arguments to `read.table` which can be used to change its behaviour.

- Setting `header=TRUE` indicates to R that the first row of the data file contains names for each of the columns.
- The argument `skip=` makes it possible to skip the specified number of lines at the top of the file.
- The argument `sep=` can be used to specify a character which separates columns.
- The argument `row.names=` specifies row names. This is either a vector of row names, a positive integer specifying which column contains the row names, or the variable name corresponding to the row.names.

## Customised Use of `read.table`

- The argument `col.names` specifies a vector of names for the columns (for the case where `header=FALSE`. The names default to `V` followed by the column number.
- The argument `as.is` controls the conversion of variables to factors (the default behaviour is to convert character variables to factors). This can either be a vector of logicals specifying whether the corresponding columns should be left unconverted, or a vector of the integer indices of the columns to be left alone.
- More flexibly, the argument `colClasses=` makes it possible to precisely specify the type of the data in the corresponding columns.

## Example

The file “mydatafile.txt” contains the (quoted) names of the islands of New Zealand bigger than 1000 square kilometers, together with the corresponding areas.

```
> nz = read.table("mydatafile.txt")
> nz
              V1     V2
1   South Island 151215
2   North Island 113729
3 Stewart Island   1746
> nz = read.table("mydatafile.txt",
                  col.names = c("Island", "Area"))
> nz
          Island   Area
1   South Island 151215
2   North Island 113729
3 Stewart Island   1746
```

## Example (Continued)

```
> nz = read.table("mydatafile.txt",
                  col.names = c("Island", "Area"),
                  row.names = "Island")
> nz
                 Area
South Island   151215
North Island   113729
Stewart Island   1746
```

## Example (Continued)

```
> nz = read.table("mydatafile.txt",
                  col.names = c("Island", "Area"),
                  row.names = "Island",
                  colClasses = c("character",
                                 "character"))
> nz
                 Area
South Island   151215
North Island   113729
Stewart Island   1746

> nz$Area
[1] "151215" "113729" "1746"
```

## Customised `read.table` Variants

There are a number of variants of `read.table` which have slightly different behaviour.

- `read.csv` arguments that (by default) columns are separated by commas.
- `read.csv2` arguments that (by default) columns are separated by semicolons and that the decimal indicator is a comma.
- `read.delim` arguments that (by default) columns are separated by tabs.
- `read.delim2` arguments that (by default) columns are separated by tabs and that the decimal indicator is a comma.

## Fixed-Width Fields

A long time ago in a galaxy far far away, people used to program in a language called Fortran and data files consisted of columns of data which were not separated by white space or commas or tabs. Instead the variables were indicated as coming from a fixed set of character positions in the file records (more specifically the columns of punched cards).

The function `read.fwf` makes it possible to read this kind of archaic data with R.

## Example

Suppose that the file `myfwffile.txt` contains the records

```
123456
987654
```

We could read this as containing there variables, each occupying two columns of the record.

```
> x = read.fwf("myfwffile.txt",
               widths = c(2, 2, 2),
               col.names = c("x", "y", "z"))
> x
   x  y  z
1 12 34 56
2 98 76 54
```

## Example (Continued)

If field widths are specified as negative, that field is skipped.

Again, using the file with records:

```
123456
987654
```

we can skip the middle field as follows.

```
> x = read.fwf("myfwffile.txt",
               widths = c(2, -2, 2),
               col.names = c("x", "z"))
> x
   x  z
1 12 56
2 98 54
```

## Printing

- We’ve seen that the `print` function provides a way of printing R objects.
- The results produced by `print` can be customised by a number of optional arguments.
- Although customisation of `print` is possible, the way in which it displays objects is relatively restricted.
- There are a number of other functions which are more flexible.

## The `cat` function

`cat` is a function which can be used to concatenate and then print character strings passed to it as arguments.

- Objects other than character strings are automatically converted to character strings by `cat`.
- By default, the strings being concatenated are separated by a space character. This can be overridden with the `sep=` argument.
- Certain sequences of characters have a special interpretation (shared with C, C++, Java and other languages). The sequence `\n` represents a *newline* character, `\t` represents a *tab* character, and `\\` represents a *backslash* character.

## Examples

Here are some simple uses of `cat`.

```
> cat("aaa", "bbb", "\n")
aaa bbb

> cat("aaa", "bbb", "\n", sep="")
aaabbb

> cat("a","b","c","d","\n",sep=" - ")
a - b - c - d -

> cat("R-squared =", R2, "\n")
R-squared = 0.7863
```

## Vector Arguments to `cat`

When the arguments to `cat` are vectors, their elements are treated as though they were separate arguments.

The system data set `letters` contains the lower case letters of the roman alphabet.

```
> letters
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
[11] "k" "l" "m" "n" "o" "p" "q" "r" "s" "t"
[21] "u" "v" "w" "x" "y" "z"

> cat("The alphabet: ",
      letters, "\n", sep = "")
The alphabet: abcdefghijklmnopqrstuvwxyz
```

## Rounding

There are a variety of S functions which help to format numbers for use with `cat`.

The function `round` rounds its argument to a specified number of decimal digits.

```
> round(runif(5), 2)
[1] 0.94 0.75 0.30 0.00 0.47
```

The function `signif` rounds its argument to a specified number of significant digits.

```
> signif(runif(5), 2)
[1] 0.54 0.23 0.73 0.25 0.91
```

## Formatting: `format`

- The R function `format` formats the elements of a vector as character strings, using a common format.
- By default, the strings are padded with spaces so that they are all the same length.
- Useful optional arguments:

| | |
|---|---|
| `width` | the (minimum) width of strings produced. |
| `trim` | if set to `TRUE` there is no padding with spaces. |
| `justify` | controls how padding takes place for strings. Takes the values `"left"`, `"right"`, `"centre"`, `"none"`. |

## Formatting Numbers with `format`

- There are a number of arguments which control the printing of numbers:

| | |
|---|---|
| `digits` | The number of digits to the right of the decimal place. |
| `scientific` | Set to `TRUE` for scientific notation and `FALSE` for standard notation. |

- The justify argument does not apply to numeric values (go figure).
- Other arguments provide finer control of how numbers are formatted.

## Format Examples

```
> format(c("a", "bb", "ccc"),
         width = 5, justify = "c")
[1] "  a  " " bb  " " ccc "

> format(1/1:5, digits = 2)
[1] "1.00" "0.50" "0.33" "0.25" "0.20"

> format(format(1/1:5, digits = 2),
         width = 6, justify = "c")
[1] " 1.00 " " 0.50 " " 0.33 " " 0.25 "
[5] " 0.20 "

> format(123456789, big.mark = ",")
[1] "123,456,789"
```

## Formatting: `sprintf`

- The `sprintf` function is modeled on the C language function of the same name.
- It provides a very flexible way of formatting vector elements as character strings.
- Inspect the R manual for full details.

## Sprintf Examples

```
> sprintf("%f", 1/3)
[1] "0.333333"

> sprintf("%e", 1/3)
[1] "3.333333e-01"

> sprintf("%.3f", 1/3)
[1] "0.333"

> sprintf("%5s", c(TRUE, FALSE))
[1] " TRUE" "FALSE"

> sprintf("%-5s", c(TRUE, FALSE))
[1] "TRUE " "FALSE"
```

## The paste Function

The paste function provides a very flexible way of pasting strings together. It is very useful when used in conjunction with cat.

paste obeys the vector recycling rule, which makes it useful for tasks like creating labels.

```
> paste("Var", 1:4)
[1] "Var 1" "Var 2" "Var 3" "Var 4"
```

In addition, paste has sep= and collapse= arguments which control how strings are glued together and make it possible to glue all the results into a single string.

```
> paste("Var", 1:4, sep="-", collapse=":")
[1] "Var-1:Var-2:Var-3:Var-4"
```

## Printing to Files

- By default, the `cat` function prints its output to the screen.
- This default can be changed using the options `file=` and `append=`.
- The `file` argument is a character string containing the name of file which is to receive the output (the default name of `""` corresponds to the terminal window).
- By default, the contents of the output file are emptied before output begins. Setting `append=TRUE` means that the output is appended to the existing file contents.

## Example: Printing a Matrix in CSV Format

Let’s consider how we might go about writing a function which makes it possible to write a numeric matrix into a file using CSV format (comma separated variables) so that it can be read into a speadsheet.

The program should be able to handle row and column names on the matrix and these should be included in the file.

The default file name should be `""`.

## Output Format

The output should have the form:

```
NNN,NNN,...,NNN
NNN,NNN,...,NNN
NNN,NNN,...,NNN
```

when there are no row or column labels and

```
,CL1,CL2,...,CLN
RL1,NNN,NNN,...,NNN
RL2,NNN,NNN,...,NNN
RL3,NNN,NNN,...,NNN
```

when there are both row and column labels. The cases with just row or column labels should also be handled.

## Quoting Strings – R Code

The row and column names are strings and so may contain commas. Because of this they must be quoted (speadsheets expect double quotes).

Quoting strings is a self-contained problem and it makes sense to write a function to carry out the task.

```
> dquote =
      function(s)
      paste("\"", s, "\"", sep = "")
```

This function is similar to the system `dQuote` function however the use of that function (and its companion `sQuote` function) is more technical than we need.

## Writing the Column Labels – Pseudo-code

We can describe the process of writing the column names as follows:

```
If there are column names {
    If there are row names
        print ","
    Print the column names quoted and separated by ","
}
```

This kind of outline sketch of what the final code will look like is known as *pseudo-code*. It is a very good idea to produce this kind of sketch when working on a program of any complexity.

## Writing the Column Labels – R Code

The pseudo-code can now be turned into R code. Because the pseudo-code contains an outline of the control-flow, it is possible to concentrate on more specific details at this point.

```
> print.colnames =
      function(x, file = "")
      if (!is.null(colnames(x))) {
          if(!is.null(rownames(x)))
              cat(",", file = file, append = TRUE)
          cat(dquote(colnames(x)), sep = ",",
              file = file, append = TRUE)
          cat("\n",
              file = file, append = TRUE)
      }
```

## Writing the Rows – Pseudo-code

The basic flow of control for printing the rows is trivial.

```
for each row {
    If there are row names
        Print this row's name followed by ","
    Print the row values, separated by ","
    Move to the next line
}
```

Again, it is possible that the row names might contain commas so they need to be quoted (with `dquote`).

## Writing the Rows – R Code

Again, translating the pseudo-code into actual R code is easy.

```
> print.rows =
      function(x, file = "")
      for(i in 1:nrow(x)) {
          if (!is.null(rownames(x)))
              cat(dquote(rownames(x)[i]), ",",
                  sep = "", file = file,
                  append = TRUE)
          cat(x[i,], sep = ",",
              file = file, append = TRUE)
          cat("\n", file = file, append = TRUE)
      }
```

## The Top-Level Function

Now that the component pieces are available it is easy to assemble them into a function.

```
> print.csv =
      function(x, file = "") {
          if (!is.matrix(x))
              stop("non-matrix argument")
          cat("", file = file)
          print.colnames(x, file = file)
          print.rows(x, file = file)
      }
```

Note that the call to `cat` in this function does *not* specify `append=TRUE`. This will cause the contents of any existing file, with name given by the `file` argument to be “zeroed out”.

## Testing The Function

```
> print.csv(VADeaths)
,"Rural Male","Rural Female","Urban Male","Urban Female"
"50-54",11.7,8.7,15.4,8.4
"55-59",18.1,11.7,24.3,13.6
"60-64",26.9,20.3,37,19.3
"65-69",41,30.9,54.6,35.1
"70-74",66,54.3,71.1,50

> colnames(VADeaths) = NULL
> print.csv(VADeaths)
"50-54",11.7,8.7,15.4,8.4
"55-59",18.1,11.7,24.3,13.6
"60-64",26.9,20.3,37,19.3
"65-69",41,30.9,54.6,35.1
"70-74",66,54.3,71.1,50
```

## Low-Level Data Input

The function `read.table` is designed for reading tabular data sets.

Sometimes it is useful to be able to read data in other formats.

There are a number of functions available for doing this.

The most useful of these are `readline`, `readlines` and `scan`.

## Obtaining a Line of Input from the User

The function `readline` prompts the user for a line of input and returns the line they type.

The argument to `readline` contains the prompt to be printed.

The result returned by `readline` is the line of text typed by the user, as a character string,

```
> cleanup =
      function()
      if(readline("Remove files? ") == "y")
      rm(list = ls())


> cleanup()
Remove files? y
```

## Menu Interactions

R has a simple function called `menu` which presents a simple menu an allows a user to choose an item from it.

```
> menu(c("Item A", "Item B", "Item C"))

1: Item A
2: Item B
3: Item C

Selection: 3
[1] 3
```

You can either type the item index or the item text to make the selection.

Typing `0` will cause the menu function to exit and return 0.

## Reading Multiple lines

The function `readLines` can be used to read multiple lines from the user terminal or from a specified file.

```
> readLines(n = 3)
first line
second line
third line
[1] "first line"  "second line" "third line"
```

A file can be specified as the first argument to `readLines`.

```
> lines = readLines("mycharfile.txt")
> length(lines)
[1] 2
```

## Reading Numeric Data with `scan`

The simplest function for reading data is `scan`.

The most basic use of scan is to read a set of white-space separated numbers from a plain text file into a vector.

```
> x = scan("myfile.txt")
Read 10 items
> x
 [1] -0.4  0.0  0.4  0.7 -1.1 -0.6  1.7  0.0
 [9] -0.7  1.7
```

The file is read item by item until the end of file is reached.

It is possible to specify that values are separated by something other than white-space using the optional argument `sep=` to specify a single character which separates items.

## Reading Other Data Types

Data types other than numeric can be read by specifying the `what=` argument to `scan`.

```
> x = scan("mycharfile.txt", what = character())
Read 13 items
> x
 [1] "Christie"   "Grimson"    "Huang"
 [4] "Kuan"       "Lauder"     "Li"
 [7] "Scaria"     "Stevenson"  "Sullivan"
[10] "Taylor"     "Vlaskovsky" "Yang"
[13] "Yip"
```

In the case of character data, white space (or other item separators) can be included within items by quoting those items with double quotes.

## Reading Data in Columns

Data of different types listed in columns in a data file can also be read with `scan`.

This is done by specifying a `what` argument which is a list whose elements have the types of the corresponding columns.

The value returned by `scan` in this case is a list containing the columns of data.

If the elements of the `what` list are named, the same names are used for the columns of data returned by `scan`.

Using `scan` is MUCH more efficient than using `read.table`.

## Example

The file “mydatafile.txt” contains the (quoted) names of the islands of New Zealand bigger than 1000 square kilometers, together with the corresponding areas.

```
> nz = scan("mydatafile.txt",
            what = list(name = character(),
                        area = numeric()))
Read 3 records
> nz
$name
[1] "South Island"   "North Island"
[3] "Stewart Island"

$area
[1] 151215 113729   1746
```

## Example (Continued)

A list returned by `scan` can be turned into a data frame by applying the function `data.frame` to it.

```
> data.frame(nz)
           name   area
1   South Island 151215
2   North Island 113729
3 Stewart Island   1746

> data.frame(nz, row.names = 1)
                 area
South Island   151215
North Island   113729
Stewart Island   1746
```

## Connections

We’ve seen functions used to read data from simple text files, but the input-output mechanism in R is much more general.

The generalised mechanism relies on an object called a *connection* which represents a location or process that data can be streamed to.

The following functions can be used to open a variety of connection types.

| | |
|---|---|
| `file` | simple text files |
| `url` | a network url |
| `gzfile` | a gzipped file |
| `bzfile` | a bzipped file |
| `unz` | a file in a zip archive |
| `pipe` | pipe to/from a process |

## Using Connections

In order to use a connection it must first be opened and it should also be closed when it is no longer needed.

The following connection opens a file connection, reads its contents and then closes it.

```
> fc = file("mycharfile.txt", "r")
> scan(fc, what = character())
Read 13 items
 [1] "Christie"   "Grimson"    "Huang"
 [4] "Kuan"       "Lauder"     "Li"
 [7] "Scaria"     "Stevenson"  "Sullivan"
[10] "Taylor"     "Vlaskovsky" "Yang"
[13] "Yip"
> close(fc)
```

## Using Connections

It is possible to do multiple reads from or writes to a connection.

The following example shows three numbers, three character strings and one line of text being read from a connection.

```
> fc = file("mixeddata.txt", "r")
> scan(fc, n = 3)
Read 3 items
[1] 10 20 30
> scan(fc, what = character(), n = 3)
Read 3 items
[1] "one"   "two"   "three"
> readLines(fc, n = 1)
[1] "11 12 13"
> close(fc)
```