READING IN DATA FROM AN EXTERNAL FILE | R LEARNING MODULES

Version info: Code for this page was tested in R version 3.0.2 (2013-09-25)
On: 2013-11-19
With: lattice 0.20-24; foreign 0.8-57; knitr 1.5

1. Reading in data from the console using the scan function

For very small data vectors it is sometimes handy to read in data directly from the prompt. This can be accomplished using the **scan** function from the command line. The **scan** function reads the fields of data in the file as specified by the **what** option, with the default being numeric. If the **what** option is specified to be **what**=character() or **what**=" " then all the fields will be read as strings. If the data are a mix of numeric, string or complex data, then a list can be used in the **what** option. The default separator for the **scan** function is any white space (single space, tab, or new line). Because the default is space delimiting, you can enter data on separate lines. When all the data have been entered, just hit the enter key twice which will terminate the scanning.

```
# Reading in numeric data
> x <- scan()
1: 3 5 6
4: 3 5 78 29
8: 34 5 1 78
12:
Read 11 items
> x
 [1]  3  5  6  3  5 78 29 34  5  1 78
> mode(x)
[1] "numeric"
# Reading in string data
# empty quotes indicates character input
> y <- scan(what=" ")
1: red blue
3: green red
5: blue yellow
7:
Read 6 items
> y
[1] "red"    "blue"   "green"  "red"    "blue"   "yellow"
> mode(y)
[1] "character"
```

2. Importing data files using the scan function

The **scan** function is an extremely flexible tool for importing data. Unlike the **read.table** function, however, which returns a data frame, the **scan** function returns a list or a vector. This makes the **scan** function less useful for inputting "rectangular" data such as the **car** data set that will been seen in later examples. In the previous example we input first numeric data and then string data directly from the console; in the following example, we input a text file. For the **what** option, we use list and then

list the variables, and after each variable, we tell R what type of variable (e.g., numeric, string) it is. In the first example, the first variable is **age**, and we tell R that **age** is a numeric variable by setting it equal to 0. The second variable is called **name**, and it is denoted as a string variable by the empty quote marks. In the second example, we list NULL first, indicating that we do not want the first variable to be read. After using the **scan** function, we use the **sapply** function, which makes a list out of a vector of names in **x**.

```
# inputting a text file and outputting a list
(x <- scan("http://stats.idre.ucla.edu/stat/data/scan.txt", what = list(age = 0,
    name = "")))
```

```
## $age
## [1] 12 24 35 20
##
## $name
## [1] "bobby"   "kate"    "david"   "michael"
```

```
# using the same text file and saving only the names as a vector
x <- scan("http://stats.idre.ucla.edu/stat/data/scan.txt", what = list(NULL, name = character()))

(x <- x[sapply(x, length) > 0])
```

```
## $name
## [1] "bobby"   "kate"    "david"   "michael"
```

```
is.vector(x)
```

```
## [1] TRUE
```

3. Reading in free formatted data from an ASCII file using the read.table function

The **read.table** function will let you read in any type of delimited ASCII file. It can read in both numeric and character values. The default is for it to read in everything as numeric data, and character data is read in as numeric, it is easiest to change that once the data has been read in using the **mode** function. This is by far the easiest and most reliable method of entering data into R.

```
# complete data, space delimited, variable names in first row
(test <- read.table("http://stats.idre.ucla.edu/stat/data/test.txt", header = TRUE))
```

```
##    prgtype gender  id ses schtyp level
## 1  general      0  70   4      1     1
## 2   vocati      1 121   4      2     1
## 3  general      0  86   4      3     1
## 4   vocati      0 141   4      3     1
## 5 academic      0 172   4      2     1
## 6 academic      0 113   4      2     1
## 7  general      0  50   3      2     1
## 8 academic      0  11   1      2     1
```

The default delimiter in **read.table** is the space delimiter, but this could create problems if there are missing data. The function will not work unless every data line has the same number of values. Thus, if there are missing data, the data lines will have different number of values, and you will receive an error. If there are missing values the easiest way to fix this problem is to change the type of delimiter. In the **read.table** function the **sep** argument is used to specify the delimiter.

```
# showing the file with missing values, space delimited (test_missing.txt data file)
prgtype  gender  id ses schtyp  level
  general    0  70    4   1      1
   vocati    1 121    4          1
  general    0  86               1
   vocati    0 141    4   3      1
 academic    0 172    4   2      1
 academic    0 113    4   2      1
  general    0  50    3   2      1
 academic    0  11    1   2      1
```

```
test.missing <- read.table("http://stats.idre.ucla.edu/stat/data/test_missing.txt",
    header = TRUE)
```

```
## Error: line 2 did not have 6 elements
```

```
# showing the file with missing data, comma delimited (test_missing_comma.txt data file)
prgtype,  gender,  id, ses, schtyp,  level
  general,    0,  70,    4,   1,      1
   vocati,    1, 121,    4,    ,      1
  general,    0,  86,     ,    ,      1
   vocati,    0, 141,    4,   3,      1
 academic,    0, 172,    4,   2,      1
 academic,    0, 113,    4,   2,      1
  general,    0,  50,    3,   2,      1
 academic,    0,  11,    1,   2,      1
```

```
(test.missing <- read.table("http://stats.idre.ucla.edu/stat/data/test_missing_comma.txt",
    header = TRUE, sep = ","))
```

```
##     prgtype gender  id ses schtyp level
## 1   general      0  70   4      1     1
## 2    vocati      1 121   4     NA     1
## 3   general      0  86  NA     NA     1
## 4    vocati      0 141   4      3     1
## 5  academic      0 172   4      2     1
## 6  academic      0 113   4      2     1
## 7   general      0  50   3      2     1
## 8  academic      0  11   1      2     1
```

The **read.table** function is very useful when reading in ASCII files that contain rectangular data. As mentioned above, the default delimiter is blank space; other delimiters must be specified by using the **sep** option and setting it equal to the delimiter in quotes (i.e., **sep=";"** for the semicolon delimited data file). Another very common type of file is the comma delimited file. The next file has been saved out of Excel as a comma delimited file. This file can be read in by the **read.table** function by using the **sep** option, but it can also be read in by the **read.csv** function which was written specifically for comma delimited files. We use the **print** function to display the contents of the object **test.csv** just to show its use.

```
test.csv <- read.csv("http://stats.idre.ucla.edu/stat/data/test.csv", header = TRUE)

print(test.csv)
```

```
##    prgtype gender  id ses schtyp level
## 1  general      0  70   4      1     1
## 2   vocati      1 121   4      2     1
## 3  general      0  86   4      3     1
## 4   vocati      0 141   4      3     1
## 5 academic      0 172   4      2     1
## 6 academic      0 113   4      2     1
## 7  general      0  50   3      2     1
## 8 academic      0  11   1      2     1
```

```
test.csv1 <- read.table("http://stats.idre.ucla.edu/stat/data/test.csv", header = TRUE,
    sep = ",")

print(test.csv1)
```

```
##    prgtype gender  id ses schtyp level
## 1  general      0  70   4      1     1
## 2   vocati      1 121   4      2     1
## 3  general      0  86   4      3     1
## 4   vocati      0 141   4      3     1
## 5 academic      0 172   4      2     1
## 6 academic      0 113   4      2     1
## 7  general      0  50   3      2     1
## 8 academic      0  11   1      2     1
```

It is, of course, also possible to use the **read.table** function for reading in files with other delimiters. The next file has semicolon delimiters, followed by a dataset that uses the letter “z” as a delimiter, both of which are acceptable delimiters in R.

```
test.semi <- read.table("http://stats.idre.ucla.edu/stat/data/testsemicolon.txt",
    header = TRUE, sep = ";")
print(test.semi)
```

```
##    prgtype gender  id ses schtyp level
## 1  general      0  70   4      1     1
## 2   vocati      1 121   4      2     1
## 3  general      0  86   4      3     1
## 4   vocati      0 141   4      3     1
## 5 academic      0 172   4      2     1
## 6 academic      0 113   4      2     1
## 7  general      0  50   3      2     1
## 8 academic      0  11   1      2     1
```

```
test.z <- read.table("http://stats.idre.ucla.edu/stat/data/testz.txt", header = TRUE,
    sep = "z")
print(test.z)
```

```
##    prgtype gender  id ses schtyp level
## 1  general      0  70   4      1     1
## 2   vocati      1 121   4      2     1
## 3  general      0  86   4      3     1
## 4   vocati      0 141   4      3     1
## 5 academic      0 172   4      2     1
## 6 academic      0 113   4      2     1
## 7  general      0  50   3      2     1
## 8 academic      0  11   1      2     1
```

4. Reading in fixed formatted files

We use the **read.fwf** function to read in data with fixed formats, and we use the **width** argument to indicate the width (number of columns) of each variable. In a fixed format file we do not have the names of the variables on the first line, and therefore they must be added after we have read in the data. We add the variable names using the **dimnames** function and the bracket notation to indicate that we are attaching names to the variables (columns) of the data file. Please note that there are several different ways to accomplish this task; this is just one of them.

```
test.fixed <- read.fwf("http://stats.idre.ucla.edu/stat/data/test_fixed.txt", width = c(8,
    1, 3, 1, 1, 1))

dimnames(test.fixed)[[2]] <- c("prgtyp", "gender", "id", "ses", "schtyp", "level")

test.fixed
```

```
##     prgtyp gender  id ses schtyp level
## 1 general      0  70   4      1     1
## 2 vocati       1 121   4      2     1
## 3 general      0  86   4      3     1
## 4 vocati       0 141   4      3     1
## 5 academic     0 172   4      2     1
## 6 academic     0 113   4      2     1
## 7 general      0  50   3      2     1
## 8 academic     0  11   1      2     1
```

For fixed format files the variables names are often in a separate file from the data. In this example the variable names are in a file called “names.txt” and the data are again in “test_fixed.txt”. This is especially convenient when the fixed format file is very large and has many variables; then it becomes rather impractical to type in all the variable names. In this situation the **width** option is used to specify the width of each variable and the **col.name** option specifies the file containing the variable names. So, first we read in the file for the names using the **scan** function. We specify that file contains character values by setting the **what** option to equal **character()**. By using the **col.names** option in the **read.fwf** function, the object **names** will supply the variables names.

```
(names <- scan("http://stats.idre.ucla.edu/stat/data/names.txt", what = character()))
```

```
## [1] "prgtyp" "gender" "id"     "ses"    "schtyp" "level"
```

```
(test.fixed <- read.fwf("http://stats.idre.ucla.edu/stat/data/test_fixed.txt",
    col.names = names, width = c(8, 1, 3, 1, 1, 1)))
```

```
##     prgtyp gender  id ses schtyp level
## 1 general       0  70   4      1     1
## 2 vocati        1 121   4      2     1
## 3 general       0  86   4      3     1
## 4 vocati        0 141   4      3     1
## 5 academic      0 172   4      2     1
## 6 academic      0 113   4      2     1
## 7 general       0  50   3      2     1
## 8 academic      0  11   1      2     1
```

5. Exporting files using the write.table function

The **write.table** function outputs data files. The first argument specifies which data frame in R is to be exported. The next argument specifies the file to be created. The default separator is a blank space but any separator can be specified in the **sep** option. The default value for both the **row.names** and **col.names** options is TRUE. In the example we specify that we do not wish to include row names. The default setting for the **quote** option is to include quotes around all the character values, i.e., around values in string variables and around the column names. As we have shown in the example it is very common not to want the quotes when creating a text file.

```
# using the test.csv data frame to write a text file with no row names and
# without quotes around the character values (both column names and string
# variables)
write.table(test.csv, file.path(tempdir(), "test1.txt"), row.names = FALSE,
    quote = FALSE)
```

6. Exporting files in Stata 6/7/8/10 format using the write.dta function

The **write.dta** function is part of the **foreign** package and writes an R data frame to a Stata data file in either Stata 6, 7, 8, or 10 format. Although these are older versions of Stata, Stata has no difficulty reading files written in older versions. It takes at least two arguments, the first one being the data frame and the second one being the output Stata data file name. If you look at the help file for **write.dta**, you will see that the function writes out a Stata 7 data file, but there are comments and options for those using later versions of Stata. In the example below, we use the **anscombe** data set that comes with R. It happens that the **anscombe** data is already a data frame, this being checked with the **is.data.frame** function.

```
library(foreign)
data(anscombe)
is.data.frame(anscombe)
```

```
## [1] TRUE
```

```
anscombe
```

```
##    x1 x2 x3 x4    y1   y2    y3    y4
## 1  10 10 10  8  8.04 9.14  7.46  6.58
## 2   8  8  8  8  6.95 8.14  6.77  5.76
## 3  13 13 13  8  7.58 8.74 12.74  7.71
```

```
## 4   9  9  9  8  8.81 8.77  7.11  8.84
## 5  11 11 11  8  8.33 9.26  7.81  8.47
## 6  14 14 14  8  9.96 8.10  8.84  7.04
## 7   6  6  6  8  7.24 6.13  6.08  5.25
## 8   4  4  4 19  4.26 3.10  5.39 12.50
## 9  12 12 12  8 10.84 9.13  8.15  5.56
## 10  7  7  7  8  4.82 7.26  6.42  7.91
## 11  5  5  5  8  5.68 4.74  5.73  6.89
```

```
write.dta(anscombe, file = file.path(tempdir(), "anscombe.dta"))
```

Now let's see an example where the data is not yet a data frame. We can use function **as.data.frame** to convert the data into a data frame. Again, these data come with R.

```
data(WorldPhones)
is.data.frame(WorldPhones)
```

```
## [1] FALSE
```

```
WorldPhones
```

```
##      N.Amer Europe Asia S.Amer Oceania Africa Mid.Amer
## 1951  45939  21574 2876   1815    1646     89      555
## 1956  60423  29990 4708   2568    2366   1411      733
## 1957  64721  32510 5230   2695    2526   1546      773
## 1958  68484  35218 6662   2845    2691   1663      836
## 1959  71799  37598 6856   3000    2868   1769      911
## 1960  76036  40341 8220   3145    3054   1905     1008
## 1961  79831  43173 9053   3338    3224   2005     1076
```

```
(phones_d <- as.data.frame(WorldPhones))
```

```
##      N.Amer Europe Asia S.Amer Oceania Africa Mid.Amer
## 1951  45939  21574 2876   1815    1646     89      555
## 1956  60423  29990 4708   2568    2366   1411      733
## 1957  64721  32510 5230   2695    2526   1546      773
## 1958  68484  35218 6662   2845    2691   1663      836
## 1959  71799  37598 6856   3000    2868   1769      911
## 1960  76036  40341 8220   3145    3054   1905     1008
## 1961  79831  43173 9053   3338    3224   2005     1076
```

```
is.data.frame(phones_d)
```

```
## [1] TRUE
```

```
write.dta(phones_d, file = file.path(tempdir(), "phones.dta"))
```

To give you an idea of what types of data can be read into R using the **foreign** package, the help file is shown below.

```
data.restore            Read an S3 Binary or data.dump File
```

```
lookup.xport            Lookup Information on a SAS XPORT Format
                        Library
read.arff               Read Data from ARFF Files
read.dbf                Read a DBF File
read.dta                Read Stata Binary Files
read.epiinfo            Read Epi Info Data Files
read.mtp                Read a Minitab Portable Worksheet
read.octave             Read Octave Text Data Files
read.spss               Read an SPSS Data File
read.ssd                Obtain a Data Frame from a SAS Permanent
                        Dataset, via read.xport
read.systat             Obtain a Data Frame from a Systat File
read.xport              Read a SAS XPORT Format Library
write.arff              Write Data into ARFF Files
write.dbf               Write a DBF File
write.dta               Write Files in Stata Binary Format
write.foreign           Write Text Files and Code to Read Them
```

Click here to report an error on this page or leave a comment

How to cite this page (http://stats.idre.ucla.edu/other/mult-pkg/faq/general/faq-how-do-i-cite-web-pages-and-programs-from-the-ucla-statistical-consulting-group/)

