Useful Tips and Commands

Lukas Arnroth

4 januari 2018

In this document some useful tricks and commands will be considered, these are commands that I use most of the time and they tend to ease the process of working with R.

Introduction to R

R is an open source statistical software, based on the functional programming language S, developed and maintained by Cran. It consists of base functions that comes with installation and user written packages. It does put much responsibility on the user in the sense that many functions that you use are written by users. You should always consult the package documentation on the Cran website when using a new package that you are unfamiliar with.

General Tips

Work in a R Project

Always work in a R project! R will automatically look for files in your project directory, rather than you having to specify the paths to all the files you want to read in. Furthermore R will automatically save plots to your project directory. You can see the files in your directory under the Files tab in the lower right hand side of the R studio window.

So when you start a new assignment do the following:

- Choose a folder for your repository (R project)
- In the R studio window go to File / New Project. Then click New Directory and then Empty Project. Once this is done, chose the location for where R will create a repository. Name the project and create.
- You should now have a folder containing a .Rproj file. Put all the code and data files associated with the assignment in this folder.

Whenever you want to open R to continue with you assignment, always open the .Rproj, rather than just opening R studios. R will have saved your global environment (something referred to as an image) and you're good to go.

Read like R

Before it becomes second nature, try to make a habit of reading your code like R does. This is done from inside out and starting on the right side of assignment. Lets look at an example

```r
x <- c(1,4,6,7)
```

This you would read as, starting on the right side, take values 1,4,6,7 and concatonate into a vector (read from inside out). Then take this object and assign it to x in the global environment. What about

```r
mean(x*10)
```

```
## [1] 45
```

So starting from inside out, take the vector x and multiply all elements in x by 10. Once this is done move outside the paranthesis and take mean. Try reading the operation below for yourself:

```r
var(c(mean(x*10), 6.5))
```

```
## [1] 741.125
```

Structure of Functions in R

For those of you not familiar with programming at all, R is a scripted functional language. You don't need to pay much attention at all as to what this means exactly, but know that everything you use in R is a function written by other people. Functions takes arguments, some necessary and some optional, and performs a task. You can use the documentation to understand what a function does. This is done by writing ? followed by the function name. In the description you find what the function(s) does. In the usage you find the parameters the function takes. If a parameter is followed by an equal sign, it means that it has a default value or setting and you don't have to supply this parameter with any object for the function to work. Parameters without any equality sign needs to be assigned an object to work. What type of objects you assign to the parameters is specified in the arguments section.

Note that if you don't use the parameter names in your own usage of the function (for example mean(object) vs. mean(x = object)) you need to supply the arguments in the same order as in the usage section of the help documentation.

If you're uncertain of how a function works and what is it's intent you should consult the examples in the bottom of the help documentation.

Write Readable Code

Try to make your code as readable as possible! I often seperate sections of code by using the comment symbol aswell. Keep the code spacious and comment on what you are doing, for your own sakes. Here's an example of how I usually seperate my code into sections

```r
######################
####### TASK 1 #######
######################

### A) Generate 100 randomstandard normals and get mean

# get 100 standard normals
x <- rnorm(100)
# calculate the mean
mean(x)

### B) Use the same values and get the variance

# calculate the variance
var(x)

######################
####### TASK 2 #######
######################

### A) Display the results as a histogram
```

```
# Show x as histogram
hist(x)
```

Google

You need to learn how to google problems that you encounter. Someone has had your problem before. One online resource which should become your best friend is stackoverflow. Just type in your problem and add stackoverflow or stack at the end of your search. You will pretty much be guaranteed to find working solutions. Just make sure to take a look at the code and be sure that it does what you think it does.

Data Types

The basic types

The fundamental variable types in R are character, numeric, integer and factor. Factor is a bit more tricky than the other three, and will be dealt with seperately. These types are not about storage, but rather the levels of the elements in your objects.

```
# character: strings
"Lukas"
```

```
## [1] "Lukas"
```

```
# integer: whole numbers
1
```

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

```
# numeric (known as double in other programming languages): real line
1.2
```

```
## [1] 1.2
```

Vectors

Vectors are the most basic type of data storage in R. This is a one dimensional object where all the elements must be of the same type. To create a vector you use c(), which techniqually reads as concatonate but I personally refer to as combine.

```
c("Lukas", "Lars", "Paulina", "Ragna", "Thomas")
```

```
## [1] "Lukas"   "Lars"    "Paulina" "Ragna"   "Thomas"
```

This is a vector of characters. If I throw in a number in the c() call, it would implicitly be converted to character. R does this conversion to the "lower" format automatically as you can't convert Lukas to a number but you can convert a number to a character.

The dimension of a vector is referred to as *length*(), which is the number of elements in the vector.

```
length(c("Lukas", "Lars", "Paulina", "Ragna", "Thomas"))
```

```
## [1] 5
```

You subset this dimension using one value within [] following the vector object. Say that I want to extract the 4th element (note that indexing in R begins at 1, so the first position is 1) I would do it as

```r
c("Lukas", "Lars", "Paulina", "Ragna", "Thomas")[4]
```

```
## [1] "Ragna"
```

If you want to transform a vector, say multiply all elements by 5, you do it as follows

```r
c(3, 5, 2) * 10
```

```
## [1] 30 50 20
```

Quicktip: if you want to create a vector of values in succession, say 1 to 10, you can use ':'

```r
c(1,2,3,4,5,6,7,8,9,10)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```r
1:10
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Matricies

A matrix has two dimensions with elements of all the same types. The dimensions are columns and rows, and in subsetting you need to specify both as $[rowid, columnid]$. If you leave an entry blank R will get all rows or columns. Also you can supply the indexing operators with vectors. Lets take a quick look.

```r
(mat <- matrix(c(1,2,3,4,5,6),
               nrow = 3, ncol = 2))
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

```r
# get row 2, col 2
mat[2, 2]
```

```
## [1] 5
```

```r
# get all rows, col 2
mat[, 2]
```

```
## [1] 4 5 6
```

```r
# get row 1 and 3, column 2
mat[c(1,3), 2]
```

```
## [1] 4 6
```

Multiplying a vector by a scalar works the same way as for the vector object. R will multiply all the elements of the matrix with this value

Data Frame

This is the way you store data in R. In many ways it is like a matrix, and you can subset is like a matrix using the [*rowid*, *colid*] mechanism. What sets it apart from a matrix is that the columns can be of different variable type. It is like a composition of vectors, in the sense that each column is a vector and that vector must be homogenous with respect to type (character, numeric etc.) There is also the $ operator for subsetting. Since we've inspected the matrix indexing, lets look at the $.

```r
(dat <- data.frame(Name = c("Mikaela", "Anders", "Regina", "Gustav"),
                   Age = c(24, 15, 57, 32),
                   # ignore the parameter below
                   stringsAsFactors = F))
```

```
##      Name Age
## 1 Mikaela  24
## 2  Anders  15
## 3  Regina  57
## 4  Gustav  32
```

```r
# can you see the similiraties with the matrix object?
```

Here we have created a dataframe with two variables, Name and Age, which are of two different types. You can subset the *dat* object using the variable names as follows

```r
dat$Name
```

```
## [1] "Mikaela" "Anders"  "Regina"  "Gustav"
```

```r
dat$Age
```

```
## [1] 24 15 57 32
```

Factors

Factors are the best way to store a nominal variable in R. The factor stores the nominal values as a vector of integers in the range $\{1, \ldots, k\}$ where k are the number of unique values. Then you have the levels of the factor, which is a vector of character strings mapped to these integers. It might seem a hassle to store what could easily be represented as characters as something this complicated, but other functions in R has unique interactions with factors. For example, plotting becomes more intuitive if you store a factor as a factor, rather than a character. A more formal presentation of the factor type variable will be presented in the Vote assignment in B4.

```r
# create character vector of names
(names <- as.character(c("Mikaela", "Anders", "Regina", "Gustav", "Anders")))
```

```
## [1] "Mikaela" "Anders"  "Regina"  "Gustav"  "Anders"
```

```r
# 4 unique values
unique(names)
```

```
## [1] "Mikaela" "Anders"  "Regina"  "Gustav"
```

```r
# convert to factor
(names <- factor(names))
```

```
## [1] Mikaela Anders  Regina  Gustav  Anders
## Levels: Anders Gustav Mikaela Regina
```

```r
# each unique name becomes a level of the factor
levels(names)
```

```
## [1] "Anders"  "Gustav"  "Mikaela" "Regina"
```

Note that R has ordered the levels in the way they first appear in the character vector. You can specify the levels yourselves in the factor creation. After the factor has been created however, changing the levels becomes more tricky as the the labels are mapped to integers, and changing the mapping changes the values as you see them. Lets to the process again, but I would like to specify the order of levels alphabetically.

```r
(names <- as.character(c("Mikaela", "Anders", "Regina", "Gustav", "Anders")))
```

```
## [1] "Mikaela" "Anders"  "Regina"  "Gustav"  "Anders"
```

```r
(names <- factor(names, levels = c("Anders", "Gustav", "Mikaela", "Regina")))
```

```
## [1] Mikaela Anders  Regina  Gustav  Anders 
## Levels: Anders Gustav Mikaela Regina
```

Subsetting

One of the more tedious tasks, at least when getting to know R, is subsetting. There is an extremely popular package called 'dplyr' which is made for easier data wrangling (data manipulation) but this package is outside the scope of any of the B-level courses. Instead we will in large focus on using the which() command.

Which()

Which() takes a object of booleans (TRUE & FALSE) and gives the positions of TRUE. Basically you supply which with some logical statement and you will get the positions of where this is fulfilled. Let's look at an example with a vector of colours. Say that you want to get the position of "red".

```r
colour <- c("red", "blue", "blue", "red", "green", "white", "blue")
colour == "red"
```

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

Using == on the vector starts a iterative process where R loops through all the elements of the colour vector and checks whether they are red or not, and returns a vector of TRUE and FALSE. If you supply which() with this vector it will give you back a vector of the positions of TRUE.

```r
which(colour == "red")
```

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

This vector of values is something you can easily use to subset the vector.

```r
colour[which(colour == "red")]
```

```
## [1] "red" "red"
```

It might seem hard to grasp the usefuleness in this example, but let's apply this on a data frame instead. Let's take a look at the well known flower iris data set to get a sense of how you can use the which() command. This dataset comes with the base package *datasets*, you can see what other datasets are available using the command $library(help = "datasets")$.

```r
str(iris)
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

If I asked you to give me the mean sepal width of setosa, how would you go about this? Using the which() command, we can start by extracting the row positions of setosa. And remember that we can use $ to subset a data frame by variable (column) names.

```r
which(iris$Species == "setosa")
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
## [24] 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46
## [47] 47 48 49 50
```

This vector contains the positions of setosa. From here on out there are many ways to go about getting the mean of sepal width. I'd prefer not to save the vector as a new object in the environment (as it quickly gets cluttered up with stuff) but you can ofcourse do that. This is how I would go about getting the mean.

```r
mean(iris$Sepal.Width[which(iris$Species == "setosa")])
```

```
## [1] 3.428
```

So first, $iris\$Sepal.Width$ extracts the sepal width column from the iris dataframe. This object is a vector.

```r
is.vector(iris$Sepal.Width)
```

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

When you want to use the square brackets for subsetting you therefore only supply one dimension to subset based on (since a vector only has length, whilst a matrix has columns and rows). So in the square brackets you supply the vector of positions where Species == "setosa".

Note that there are many logical operators, apart from '=='. Examples are <, >, <=, >=, != etc.

Sidenote: multiple logical conditions.

Say that you wanted to get the mean of sepal width for both virginica and setosa, how could you go about that? Well, since you have three types of species you could simple use the not equal operator and find positions of != "versicolor". But if there where more than three species the most efficient way would be to use multiple logical statements. For this you use |,& which are read as OR and AND respectively. Note that if you use only one symbol you will compare vectors element by element. And using double symbols (||, &&) will return a vector of length one, which means that it is used when comparing two or more length one vectors. Lets look at all four ways and try to figure out which one is the correct

```r
which(iris$Species == "setosa" & iris$Species == "virginica")
```

```
## integer(0)
```

```r
which(iris$Species == "setosa" && iris$Species == "virginica")
```

```
## integer(0)
```

```r
which(iris$Species == "setosa" | iris$Species == "virginica")
```

```
##   [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17
##  [18]  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34
##  [35]  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50 101
##  [52] 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118
##  [69] 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135
##  [86] 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150
```

```r
which(iris$Species == "setosa" || iris$Species == "virginica")
```

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

Well, for starters, using the double versions is not correct. I want a vector returned where checks has been made element wise. Secondly & is not correct. It will never evaluate to true as no entry can be both setosa and virginica. So | is correct. Just to give you an idea of how || is used, you can use it within a for loop to get the same result

```
vec <- NULL
for(i in 1:nrow(iris)){
  if((iris$Species[i] == "setosa") || (iris$Species[i] == "virginica") == TRUE){
    vec[i] <- TRUE
  } else {
    vec[i] <- FALSE
  }
}

which(vec)
```

```
##   [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17
##  [18]  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34
##  [35]  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50 101
##  [52] 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118
##  [69] 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135
##  [86] 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150
```

Using the | and & can be treacherous as it won't throw an error if the vectors are of unequal length. If you think you are comparing two vectors of equal length, but one is 2 elements less, it will cycle through the first two values of the smaller vector, comparing it to the last two of the larger.

unique()

Although not only useful in the context of subsetting I will present this function for usage in this context. Let's say that you are uncertain of all the different types of species in the iris dataset. How would you go about inspecting the unique values?

```
unique(iris$Species)
```

```
## [1] setosa     versicolor virginica
## Levels: setosa versicolor virginica
```

So, three types! You can use this if you feel uncertain about the spelling of a level of the factor.

```
which(iris$Species == unique(iris$Species)[1])
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
## [24] 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46
## [47] 47 48 49 50
```

```
# equivalent to:
# which(iris$Species == setosa)
```

Do not use the unique() command on numeric and integer vectors with many values, it will not return anything useful. Generally it will return as many values as there are elements in the vector.

Column Names and Positions

One common act in the beginning of your R careers is to look at the dataset in the global environment and to check for a variable name or to, as I often did, count which position a variable name has. As you might recall you can use the square brackets to subset a dataframe the same way you would a matrix, [$rowindex$, $columnindex$].

The colnames() function returns the variable names of a dataframe.

```
colnames(iris)
```

```
## [1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"
## [5] "Species"
```

You can use this to subset the dataframe should you want. Say that you want the third element of columnnames

```
iris[, colnames(iris)[3]]
```

```
##   [1] 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 1.5 1.6 1.4 1.1 1.2 1.5 1.3
##  [18] 1.4 1.7 1.5 1.7 1.5 1.0 1.7 1.9 1.6 1.6 1.5 1.4 1.6 1.6 1.5 1.5 1.4
##  [35] 1.5 1.2 1.3 1.4 1.3 1.5 1.3 1.3 1.3 1.6 1.9 1.4 1.6 1.4 1.5 1.4 4.7
##  [52] 4.5 4.9 4.0 4.6 4.5 4.7 3.3 4.6 3.9 3.5 4.2 4.0 4.7 3.6 4.4 4.5 4.1
##  [69] 4.5 3.9 4.8 4.0 4.9 4.7 4.3 4.4 4.8 5.0 4.5 3.5 3.8 3.7 3.9 5.1 4.5
##  [86] 4.5 4.7 4.4 4.1 4.0 4.4 4.6 4.0 3.3 4.2 4.2 4.2 4.3 3.0 4.1 6.0 5.1
## [103] 5.9 5.6 5.8 6.6 4.5 6.3 5.8 6.1 5.1 5.3 5.5 5.0 5.1 5.3 5.5 6.7 6.9
## [120] 5.0 5.7 4.9 6.7 4.9 5.7 6.0 4.8 4.9 5.6 5.8 6.1 6.4 5.6 5.1 5.6 6.1
## [137] 5.6 5.5 4.8 5.4 5.6 5.1 5.1 5.9 5.7 5.2 5.0 5.2 5.4 5.1
```

Techniqually I'm counting at the moment, but it's much easier to look at the vector of column names rather than opening the data into the reader or looking in the global environment. You want to spend as little time as possible doing manual tasks!

Working with Data

The .csv Format

Ofcourse excel files are used but it in general it surpassed vastly in usage by the .csv format. For example, data from SQL databases will often be saved to the .csv format. Csv stands for comma seperated value. These files are text files where the data points are most often seperated by commas (but often other symbols, such as tab and whitespace) and a new observation is assumed to be given by a new line.

read.csv()

The most commonly used function for reading in .csv data files is the read.csv() function. In my working directory I have a text file named $japanese_emmigration.txt$ which contains data on japenese emmigration during the 20th century. Let's try loading the data using all the default settings of the read.csv() function and see what happens

```
(dat <- read.csv("japanese_emmigration.txt"))
```

```
##                     X1.Y.X2.X3.X4.X5.X6
## 1          Aomori;59;49;16.6;164.8;0;0
## 2            Akita;8;50;14.8;162.8;0;0
## 3           Iwate;55;29;14.7;158.9;0;0
## 4       Miyagi;51;39;17.0;130.9;0.01;0
## 5         Yamagata;32;43;8.4;144.7;0;0
## 6      Fukushima;173;27;17.3;135.0;0;1
## 7      Tochigi;52;34;10.9;116.5;0.02;0
## 8         Ibaraki;38;42;16.2;119.1;0;0
## 9          Gunma;53;39;12.8;90.5;0.1;0
## 10           Saitama;15;46;0.1;94.8;0;0
## 11         Chiba;47;47;9.1;108.1;0.85;0
## 12         Tokyo;110;44;3.1;95.6;0.02;1
## 13     Kanagawa;277;47;1.1;91.4;2.26;1
## 14      Niigata;84;51;4.7;126.3;5.14;0
## 15          Toyama;113;56;5.2;109.5;0;0
## 16     Ishikawa;55;37;-2.6;99.6;0.28;0
## 17          Nagano;286;43;11.4;78.2;0;0
## 18              Gifu;39;47;7.1;68.5;0;0
## 19      Yamanashi;233;55;10.7;69.8;0;1
## 20    Shizuoka;152;49;10.3;77.7;0.11;1
## 21             Aichi;73;49;2.8;70.7;0;0
## 22           Fukui;148;44;10.9;77.8;0;0
## 23        Shiga;382;45;5.3;78.3;0.81;0
## 24             Kyoto;49;42;3.9;74.2;0;0
## 25           Mie;70;45;11.8;79.9;0.14;0
## 26             Nara;120;46;1.3;67.5;0;1
## 27             Hyogo;56;51;8.3;69.6;0;1
## 28            Osaka;72;64;23.2;70.8;0;0
## 29     Wakayama;557;42;6.4;57.4;0.55;1
## 30          Tottori;242;56;8.7;83.5;0;0
## 31     Okayama;746;49;10.5;72.2;0.62;1
## 32           Shimane;61;52;4.7;83.5;0;0
## 33  Hiroshima;895;42;9.1;53.9;111.24;0
## 34 Yamaguchi;668;50;10.8;93.3;104.24;1
## 35           Kagawa;107;70;4.1;54.0;0;0
## 36           Ehime;322;44;-5.2;76.8;0;1
## 37        Tokushima;139;43;4.4;73.3;0;0
## 38           Kochi;152;30;8.7;102.7;0;0
## 39      Fukuoka;242;53;6.1;95.3;21.8;0
## 40              Saga;78;40;4.7;98.8;0;0
## 41             Oita;58;40;18.2;78.7;0;0
## 42          Nagasaki;49;40;3.0;76.5;0;0
## 43  Kumamoto;584;41;-3.2;111.5;42.47;1
## 44        Miyazaki;52;33;3.1;137.5;0;0
## 45    Kagoshima;188;34;6.1;116.1;0.2;1
```

```r
# 45 rows, 1 column
nrow(dat);ncol(dat)
```

```
## [1] 45
```

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

That does not look right at all! Using the default settings of read.csv() means that R is looking for ‘,’ as value seperator, but my data has ‘;’ as value seperator. It still has new line as seperator between observations. The results is that my data import gives me 45 observations of only one variable, which is a string containing all 7 variables in my data. Lets try this again, but manually specifying the seperator as ‘;’.

```r
(dat <- read.csv("japanese_emmigration.txt", sep = ";"))
```

```
##            X1   Y X2   X3    X4     X5 X6
## 1      Aomori  59 49 16.6 164.8   0.00  0
## 2       Akita   8 50 14.8 162.8   0.00  0
## 3       Iwate  55 29 14.7 158.9   0.00  0
## 4     Miyagi  51 39 17.0 130.9   0.01  0
## 5   Yamagata  32 43  8.4 144.7   0.00  0
## 6  Fukushima 173 27 17.3 135.0   0.00  1
## 7    Tochigi  52 34 10.9 116.5   0.02  0
## 8    Ibaraki  38 42 16.2 119.1   0.00  0
## 9      Gunma  53 39 12.8  90.5   0.10  0
## 10   Saitama  15 46  0.1  94.8   0.00  0
## 11     Chiba  47 47  9.1 108.1   0.85  0
## 12     Tokyo 110 44  3.1  95.6   0.02  1
## 13  Kanagawa 277 47  1.1  91.4   2.26  1
## 14   Niigata  84 51  4.7 126.3   5.14  0
## 15    Toyama 113 56  5.2 109.5   0.00  0
## 16  Ishikawa  55 37 -2.6  99.6   0.28  0
## 17    Nagano 286 43 11.4  78.2   0.00  0
## 18      Gifu  39 47  7.1  68.5   0.00  0
## 19 Yamanashi 233 55 10.7  69.8   0.00  1
## 20  Shizuoka 152 49 10.3  77.7   0.11  1
## 21     Aichi  73 49  2.8  70.7   0.00  0
## 22     Fukui 148 44 10.9  77.8   0.00  0
## 23     Shiga 382 45  5.3  78.3   0.81  0
## 24     Kyoto  49 42  3.9  74.2   0.00  0
## 25       Mie  70 45 11.8  79.9   0.14  0
## 26      Nara 120 46  1.3  67.5   0.00  1
## 27     Hyogo  56 51  8.3  69.6   0.00  1
## 28     Osaka  72 64 23.2  70.8   0.00  0
## 29  Wakayama 557 42  6.4  57.4   0.55  1
## 30   Tottori 242 56  8.7  83.5   0.00  0
## 31   Okayama 746 49 10.5  72.2   0.62  1
## 32   Shimane  61 52  4.7  83.5   0.00  0
## 33 Hiroshima 895 42  9.1  53.9 111.24  0
## 34 Yamaguchi 668 50 10.8  93.3 104.24  1
## 35    Kagawa 107 70  4.1  54.0   0.00  0
## 36     Ehime 322 44 -5.2  76.8   0.00  1
## 37 Tokushima 139 43  4.4  73.3   0.00  0
## 38     Kochi 152 30  8.7 102.7   0.00  0
## 39   Fukuoka 242 53  6.1  95.3  21.80  0
## 40      Saga  78 40  4.7  98.8   0.00  0
## 41      Oita  58 40 18.2  78.7   0.00  0
## 42  Nagasaki  49 40  3.0  76.5   0.00  0
## 43  Kumamoto 584 41 -3.2 111.5  42.47  1
## 44  Miyazaki  52 33  3.1 137.5   0.00  0
## 45 Kagoshima 188 34  6.1 116.1   0.20  1
```

```r
# 45 rows, 7 columns
nrow(dat);ncol(dat)
```

```
## [1] 45

## [1] 7
```

In general, I recommend that you open the file in your operating systems default text editor and look at what character is seperating the values. Note that if it is tab seperated you use '\t' to represent it!

Note that read.csv creates a dataframe in the global environment. On a side note, you could see the dataframe as a dynamic matrix of sorts. Matricies in R needs to contain the same type of objects in all positions (character, numeric etc.) whilst a data frame can have different type of objects in the column. The dataframe can be said to be homogenous with respect to the columns.

```
is.data.frame(dat)
```

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

Variable Names

The variable names of my data is not informative to anyone else but me however. This is something you should fix directly! This is easily done using the colnames() function. Simply assign a vector of characters, the same length as the number of columns of the data frame, to the colnames() of the dataframe.

```
(colnames(dat) <- c("prefecture","emigrants","cultivated","farmland",
                    "arable","labors","pioneer"))
```

```
## [1] "prefecture" "emigrants"  "cultivated" "farmland"   "arable"
## [6] "labors"     "pioneer"
```

str() & summary()

The str() function, standing for structure, is a good first step after you have imported the data to your satisfaction. This gives you a overview of your data.

```
str(dat)
```

```
## 'data.frame':    45 obs. of  7 variables:
##  $ prefecture: Factor w/ 45 levels "Aichi","Akita",..: 3 2 15 23 43 8 37 13 10 33 ...
##  $ emigrants : int  59 8 55 51 32 173 52 38 53 15 ...
##  $ cultivated: int  49 50 29 39 43 27 34 42 39 46 ...
##  $ farmland  : num  16.6 14.8 14.7 17 8.4 17.3 10.9 16.2 12.8 0.1 ...
##  $ arable    : num  165 163 159 131 145 ...
##  $ labors    : num  0 0 0 0.01 0 0 0.02 0 0.1 0 ...
##  $ pioneer   : int  0 0 0 0 0 1 0 0 0 0 ...
```

They way I mainly use this function is to get a sense of whether all the variables has been interpreted correctly by R. For example, could pioneer by stored in a different manner than as an integer? How many unique values does that variable have?

```
unique(dat$pioneer)
```

```
## [1] 0 1
```

So it is a binary variable. Perhaps it could better be stored as a factor? This would at least give more interpretable plots later.

```
dat$pioneer <- as.factor(dat$pioneer)
levels(dat$pioneer) <- c("No pioneer", "Pioneer")
str(dat); dat$pioneer
```

```
## 'data.frame':    45 obs. of  7 variables:
##  $ prefecture: Factor w/ 45 levels "Aichi","Akita",..: 3 2 15 23 43 8 37 13 10 33 ...
##  $ emigrants : int  59 8 55 51 32 173 52 38 53 15 ...
##  $ cultivated: int  49 50 29 39 43 27 34 42 39 46 ...
##  $ farmland  : num  16.6 14.8 14.7 17 8.4 17.3 10.9 16.2 12.8 0.1 ...
##  $ arable    : num  165 163 159 131 145 ...
##  $ labors    : num  0 0 0 0.01 0 0 0.02 0 0.1 0 ...
##  $ pioneer   : Factor w/ 2 levels "No pioneer","Pioneer": 1 1 1 1 1 2 1 1 1 1 ...
##  [1] No pioneer No pioneer No pioneer No pioneer No pioneer Pioneer
##  [7] No pioneer No pioneer No pioneer No pioneer No pioneer Pioneer
## [13] Pioneer    No pioneer No pioneer No pioneer No pioneer No pioneer
## [19] Pioneer    Pioneer    No pioneer No pioneer No pioneer No pioneer
## [25] No pioneer Pioneer    Pioneer    No pioneer Pioneer    No pioneer
## [31] Pioneer    No pioneer No pioneer Pioneer    No pioneer Pioneer
## [37] No pioneer No pioneer No pioneer No pioneer No pioneer No pioneer
## [43] Pioneer    No pioneer Pioneer
## Levels: No pioneer Pioneer
```

Another reason for creating this factor with more informative values than 0 and 1 is that you don't forget what 0 and 1 stands for!