R literacy for digital soil mapping. Part 5

Soil Security Laboratory

2017

1 Manipulating data

1.1 Modes, classes, attributes, length, and coercion

As described before, the mode of an object describes the type of data that it contains. In R, mode is an object attribute. All objects have at least two attributes: mode and length, but may objects have more.

```
x <- 1:10
mode(x)
```

```
## [1] "numeric"
```

```
length(x)
```

```
## [1] 10
```

It is often necessary to change the mode of a data structure, e.g., to have your data displayed differently, or to apply a function that only works with a particular type of data structure. In R this is called coercion. There are many functions in R that have the structure `as.something` that change the mode of a submitted object to "something". For example, say you want to treat numeric data as character data.

```
x <- 1:10
as.character(x)
```

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

Or, you may want to turn a matrix into a data frame.

```
X <- matrix(1:30, nrow = 3)
as.data.frame(X)
```

```
##   V1 V2 V3 V4 V5 V6 V7 V8 V9 V10
## 1  1  4  7 10 13 16 19 22 25  28
## 2  2  5  8 11 14 17 20 23 26  29
## 3  3  6  9 12 15 18 21 24 27  30
```

If you are unsure of whether or not a coercion function exists, give it a try—two other common examples are `as.numeric` and `as.vector`.

Attributes are important internally for determining how objects should be handled by various functions. In particular, the `class` attribute determines how a particular object will be handled by a given function. For example, output from a linear regression has the class "`lm`" and will be handled differently by the `print` function than will a data frame, which has the class "`data.frame`". The utility of this object-orientated approach will become more apparent later on.

It is often necessary to know the length of an object. Of course, length can mean different things. Three useful functions for this are `nrow`, `NROW`, and `length`.

The function `nrow` will return the number of rows in a two-dimensional data structure.

```
X <- matrix(1:30, nrow = 3)
X
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]    1    4    7   10   13   16   19   22   25    28
## [2,]    2    5    8   11   14   17   20   23   26    29
## [3,]    3    6    9   12   15   18   21   24   27    30
```

```
nrow(X)
```

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

The vertical analog is `ncol`.

```
ncol(X)
```

```
## [1] 10
```

You can get both of these at once with the `dim` function.

```
dim(X)
```

```
## [1]  3 10
```

For a vector, use the function `NROW` or `length`.

```
x <- 1:10
NROW(x)
```

```
## [1] 10
```

The value returned from the function `length` depends on the type of data structure you submit, but for most data structures, it is the total number of elements.

```
length(X)
```

```
## [1] 30
```

```
length(x)
```

```
## [1] 10
```

1.2 Indexing, sub-setting, sorting, and locating data

Sub-setting and indexing are ways to select specific parts of the data structure (such as specific rows within a data frame) within R. Indexing (also know as sub-scripting) is done using the square braces in R:

```
v1 <- c(5, 1, 3, 8)
v1
```

```
## [1] 5 1 3 8
```

```
v1[3]
```

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

R is very flexible in terms of what can be selected or excluded. For example, the following returns the 1^{st} through 3^{rd} observation:

```
v1[1:3]
```

```
## [1] 5 1 3
```

While this returns all but the 4^{th} observation:

```
v1[-4]
```

```
## [1] 5 1 3
```

This bracket notation can also be used with relational constraints. For example, if we want only those observations that are <5.0:

```
v1[v1 < 5]
```

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

This may seem confusing, but if we evaluate each piece separately, it becomes more clear:

```
v1 < 5
```

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

```
v1[c(FALSE, TRUE, TRUE, FALSE)]
```

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

While we are on the topic of subscripts, we should noted that, unlike some other programming languages, the size of a vector in R is not limited by its initial assignment. This is true for other data structures as well. To increase the size of a vector, just assign a value to a position that does not currently exist:

```
length(v1)
```

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

```
v1[8] <- 10
length(v1)
```

```
## [1] 8
```

```
v1
```

```
## [1]  5  1  3  8 NA NA NA 10
```

Indexing can be applied to other data structures in a similar manner as shown above. For data frames and matrices, however, we are now working with two dimensions. In specifying indices, row numbers are given first. We will use our `soil.data` set to illustrate the following few examples:

```
library(ithir)
data(USYD_soil1)
soil.data <- USYD_soil1
dim(soil.data)
```

```
## [1] 166  16
```

```
str(soil.data)
```

```
## 'data.frame': 166 obs. of  16 variables:
##  $ PROFILE     : int  1 1 1 1 1 1 2 2 2 2 ...
##  $ Landclass   : Factor w/ 4 levels "Cropping","Forest",..: 4 4 4 4 4 4 3 3 3 3 ...
##  $ Upper.Depth : num  0 0.02 0.05 0.1 0.2 0.7 0 0.02 0.05 0.1 ...
##  $ Lower.Depth : num  0.02 0.05 0.1 0.2 0.3 0.8 0.02 0.05 0.1 0.2 ...
##  $ clay        : int  8 8 8 8 NA 57 9 9 9 NA ...
##  $ silt        : int  9 9 10 10 10 8 10 10 10 10 ...
##  $ sand        : int  83 83 82 83 79 36 81 80 80 81 ...
##  $ pH_CaCl2    : num  6.35 6.34 4.76 4.51 4.64 6.49 5.91 5.94 5.63 4.22 ...
##  $ Total_Carbon: num  1.07 0.98 0.73 0.39 0.23 0.35 1.14 1.14 1.01 0.48 ...
##  $ EC          : num  0.168 0.137 0.072 0.034 NA 0.059 0.123 0.101 0.026 0.042 ...
##  $ ESP         : num  0.3 0.5 0.9 0.2 0.9 0.3 0.3 0.6 NA NA ...
##  $ ExchNa      : num  0.01 0.02 0.02 0 0.02 0.04 0.01 0.02 NA NA ...
##  $ ExchK       : num  0.71 0.47 0.52 0.38 0.43 0.46 0.7 0.56 NA NA ...
##  $ ExchCa      : num  3.17 3.5 1.34 1.03 1.5 9.13 2.92 3.2 NA NA ...
##  $ ExchMg      : num  0.59 0.6 0.22 0.22 0.5 5.02 0.51 0.5 NA NA ...
##  $ CEC         : num  5.29 3.7 2.86 2.92 2.6 ...
```

If we want to subset out only the first 5 rows, and the first 2 columns:

```
soil.data[1:5, 1:2]
```

```
##   PROFILE      Landclass
## 1       1 native pasture
## 2       1 native pasture
## 3       1 native pasture
## 4       1 native pasture
## 5       1 native pasture
```

If an index is left out, `R` returns all values in that dimension (you need to include the comma).

```
soil.data[1:2, ]
```

```
##   PROFILE      Landclass Upper.Depth Lower.Depth clay silt sand pH_CaCl2
## 1       1 native pasture        0.00        0.02    8    9   83     6.35
```

```
## 2       1 native pasture        0.02        0.05    8    9   83      6.34
##   Total_Carbon    EC ESP ExchNa ExchK ExchCa ExchMg  CEC
## 1         1.07 0.168 0.3   0.01  0.71   3.17   0.59 5.29
## 2         0.98 0.137 0.5   0.02  0.47   3.50   0.60 3.70
```

You can also specify row or column names directly within the brackets—this can be very handy when column order may change in future versions of your code.

```
soil.data[1:5, "Total_Carbon"]
```

```
## [1] 1.07 0.98 0.73 0.39 0.23
```

You can also specify multiple column names using the `c` function.

```
soil.data[1:5, c("Total_Carbon", "CEC")]
```

```
##   Total_Carbon  CEC
## 1         1.07 5.29
## 2         0.98 3.70
## 3         0.73 2.86
## 4         0.39 2.92
## 5         0.23 2.60
```

Relational constraints can also be used in indexes. Lets subset out the soil observations that are extremely sodic i.e an ESP greater than 10%.

```
na.omit(soil.data[soil.data$ESP > 10, ])
```

```
##     PROFILE      Landclass Upper.Depth Lower.Depth clay silt sand pH_CaCl2
## 68       12         Forest        0.02        0.05   22   13   64     4.65
## 111      19 native pasture        0.16        0.26   32   13   56     6.10
## 113      20       Cropping        0.00        0.02    9    9   81     4.64
## 117      20       Cropping        0.30        0.40   37    7   56     6.30
## 118      20       Cropping        0.70        0.80   20   14   67     7.17
## 123      21       Cropping        0.25        0.35   25   16   59     5.05
## 147      26       Cropping        0.15        0.24   51    8   42     6.27
## 148      26       Cropping        0.70        0.80   50    9   40     7.81
##     Total_Carbon    EC  ESP ExchNa ExchK ExchCa ExchMg   CEC
## 68          1.49 0.499 13.0   1.00  0.74   2.17   3.76  6.85
## 111         0.50 0.223 17.4   2.62  0.30   3.74   8.37 11.97
## 113         1.08 0.301 11.1   0.31  0.87   1.01   0.63  3.02
## 117         0.32 0.214 12.1   1.66  0.27   4.19   7.61 12.44
## 118         0.09 0.292 21.2   2.88  0.33   2.86   7.50 10.23
## 123         0.25 0.073 13.2   0.95  0.30   2.00   3.92  5.29
## 147         0.63 0.134 10.4   2.09  0.74   5.09  12.23 14.29
## 148         0.84 0.820 16.4   4.93  0.91   7.52  16.72 23.34
```

While indexing can clearly be used to create a subset of data that meet certain criteria, the `subset` function is often easier and shorter to use for data frames. Sub-setting is used to select a subset of a vector, data frame, or matrix that meets a certain criterion (or criteria). To return what was given in the last example.

```
subset(soil.data, ESP > 10)
```

```
##     PROFILE      Landclass Upper.Depth Lower.Depth clay silt sand pH_CaCl2
## 68       12         Forest        0.02        0.05   22   13   64     4.65
## 111      19 native pasture        0.16        0.26   32   13   56     6.10
## 113      20       Cropping        0.00        0.02    9    9   81     4.64
## 117      20       Cropping        0.30        0.40   37    7   56     6.30
## 118      20       Cropping        0.70        0.80   20   14   67     7.17
## 123      21       Cropping        0.25        0.35   25   16   59     5.05
## 147      26       Cropping        0.15        0.24   51    8   42     6.27
## 148      26       Cropping        0.70        0.80   50    9   40     7.81
##     Total_Carbon    EC  ESP ExchNa ExchK ExchCa ExchMg   CEC
## 68          1.49 0.499 13.0   1.00  0.74   2.17   3.76  6.85
## 111         0.50 0.223 17.4   2.62  0.30   3.74   8.37 11.97
## 113         1.08 0.301 11.1   0.31  0.87   1.01   0.63  3.02
## 117         0.32 0.214 12.1   1.66  0.27   4.19   7.61 12.44
## 118         0.09 0.292 21.2   2.88  0.33   2.86   7.50 10.23
## 123         0.25 0.073 13.2   0.95  0.30   2.00   3.92  5.29
## 147         0.63 0.134 10.4   2.09  0.74   5.09  12.23 14.29
## 148         0.84 0.820 16.4   4.93  0.91   7.52  16.72 23.34
```

Note that the `$` notation does not need to be used in the `subset` function, As with indexing multiple constraints can also be used:

```
subset(soil.data, ESP > 10 & Lower.Depth > 0.3)
```

```
##     PROFILE Landclass Upper.Depth Lower.Depth clay silt sand pH_CaCl2
## 117      20  Cropping        0.30        0.40   37    7   56     6.30
## 118      20  Cropping        0.70        0.80   20   14   67     7.17
## 123      21  Cropping        0.25        0.35   25   16   59     5.05
## 148      26  Cropping        0.70        0.80   50    9   40     7.81
##     Total_Carbon    EC  ESP ExchNa ExchK ExchCa ExchMg   CEC
## 117         0.32 0.214 12.1   1.66  0.27   4.19   7.61 12.44
## 118         0.09 0.292 21.2   2.88  0.33   2.86   7.50 10.23
## 123         0.25 0.073 13.2   0.95  0.30   2.00   3.92  5.29
## 148         0.84 0.820 16.4   4.93  0.91   7.52  16.72 23.34
```

In some cases you may want to select observations that include any one value out of a set of possibilities. Say we only want those observations where `Landclass` is `native pasture` or `forest`. We could use:

```
subset(soil.data, Landclass == "Forest" | Landclass == "native pasture")
```

But, this is an easier way (we are using the `head` function just to limit the number of outputted rows. So try it without the `head` function).

```
head(subset(soil.data, Landclass %in% c("Forest", "native pasture")))
```

```
##   PROFILE      Landclass Upper.Depth Lower.Depth clay silt sand pH_CaCl2
## 1       1 native pasture        0.00        0.02    8    9   83     6.35
## 2       1 native pasture        0.02        0.05    8    9   83     6.34
## 3       1 native pasture        0.05        0.10    8   10   82     4.76
## 4       1 native pasture        0.10        0.20    8   10   83     4.51
```

```
## 5       1 native pasture        0.20         0.30   NA   10   79      4.64
## 6       1 native pasture        0.70         0.80   57    8   36      6.49
##   Total_Carbon    EC ESP ExchNa ExchK ExchCa ExchMg   CEC
## 1         1.07 0.168 0.3   0.01  0.71   3.17   0.59  5.29
## 2         0.98 0.137 0.5   0.02  0.47   3.50   0.60  3.70
## 3         0.73 0.072 0.9   0.02  0.52   1.34   0.22  2.86
## 4         0.39 0.034 0.2   0.00  0.38   1.03   0.22  2.92
## 5         0.23    NA 0.9   0.02  0.43   1.50   0.50  2.60
## 6         0.35 0.059 0.3   0.04  0.46   9.13   5.02 14.96
```

Both of the above methods produce the same result, so it just comes down to a matter of efficiency.

Indexing matrices and arrays follows what we have just covered. For example:

```
X <- matrix(1:30, nrow = 3)
X
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]    1    4    7   10   13   16   19   22   25    28
## [2,]    2    5    8   11   14   17   20   23   26    29
## [3,]    3    6    9   12   15   18   21   24   27    30
```

```
X[3, 8]
```

```
## [1] 24
```

```
X[, 3]
```

```
## [1] 7 8 9
```

```
Y <- array(1:90, dim = c(3, 10, 3))
Y[3, 1, 1]
```

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

Indexing is a little trickier for lists—you need to use double square braces, `[[i]]`, to specify an element within a list. Of course, if the element within the list has multiple elements, you could use indexing to select specific elements within it.

```
list.1 <- list(1:10, X, Y)
list.1[[1]]
```

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

It is also possible to use double, triple, etc. indexing with all types of data structures. `R` evaluates the expression from left to right. As a simple example, lets extract the element on the third row of the second column of the second element of `list.1`:

```
list.1[[2]][3, 2]
```

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

An easy way to divide data into groups is to use the `split` function. This function will divide a data structure (typically a vector or a data frame) into one subset for each level of the variable you would like to split by. The subsets are stored together in a list. Here we split our `soil.data` set into the separate or individual soil profile (splitting by the `PROFILE` column—note output is not shown here for sake of brevity).

```
soil.data.split <- split(soil.data, soil.data$PROFILE)
```

If you apply `split` to individual vectors, the resulting list can be used directly in some plotting or summarizing functions to give you results for each separate group. (There are usually other ways to arrive at this type of result). The `split` function can also be handy for manipulating and analyzing data by some grouping variable, as we will see later.

It is often necessary to sort data. For a single vector, this is done with the function `sort`.

```
x <- rnorm(5)
x

## [1] -0.8709296  1.1881006 -1.7375377  0.5447195 -0.2273537

y <- sort(x)
y

## [1] -1.7375377 -0.8709296 -0.2273537  0.5447195  1.1881006
```

But what if you want to sort an entire data frame by one column? In this case it is necessary to use the function `order`, in combination with indexing.

```
head(soil.data[order(soil.data$clay), ])

##     PROFILE        Landclass Upper.Depth Lower.Depth clay silt sand
## 116      20         Cropping        0.10        0.30    5   11   85
## 1         1   native pasture        0.00        0.02    8    9   83
## 2         1   native pasture        0.02        0.05    8    9   83
## 3         1   native pasture        0.05        0.10    8   10   82
## 4         1   native pasture        0.10        0.20    8   10   83
## 7         2 improved pasture        0.00        0.02    9   10   81
##     pH_CaCl2 Total_Carbon    EC ESP ExchNa ExchK ExchCa ExchMg  CEC
## 116     5.33         0.24 0.033 4.0   0.10  0.18   1.40   0.70 1.90
## 1       6.35         1.07 0.168 0.3   0.01  0.71   3.17   0.59 5.29
## 2       6.34         0.98 0.137 0.5   0.02  0.47   3.50   0.60 3.70
## 3       4.76         0.73 0.072 0.9   0.02  0.52   1.34   0.22 2.86
## 4       4.51         0.39 0.034 0.2   0.00  0.38   1.03   0.22 2.92
## 7       5.91         1.14 0.123 0.3   0.01  0.70   2.92   0.51 3.59
```

The function `order` returns a vector that contains the row positions of the ranked data:

```
order(soil.data$clay)

##   [1] 116   1   2   3   4   7   8   9  16 113 114 115  14  31  32  61  62
##  [18]  64  45 126 149 150  37  38  39 101 102 103 107 108 151  34  42  44
```

```
## [35]  77  13  43  46  50 109 125  78 144  35  51  79  96 110 119 120 121
## [52] 127 134  67  97 130 135 145 161  36  98 136 140 146 160 162  17  52
## [69]  83 104 118 131 139 141  65  84  85  68  69  89  99  53 156  72 123
## [86] 137  54  56  80  95 105  57  90  74 132  58  81  91 128  20  55 111
## [103] 142  86 163 106 154  11 129  18  87 112 117  25  75  21 155  66  92
## [120]  22  26 152 164 133 138  47  71  41  59  93  60 148  24 147  12 158
## [137] 165   6  82 166  48  88 159  28  29  94  76 100  40   5  10  15  19
## [154]  23  27  30  33  49  63  70  73 122 124 143 153 157
```

The previous discussion in this section showed how to isolate data that meet certain criteria from a data structure. But sometimes it is important to know where data resides in its original data structure. But sometimes it is important to know where data resides in its original data structure. To functions that are handy for locating data within an `R` data structure are **`match`** and **`which`**. The **`match`** function will tell you where specific values reside in a data structure, while the **`which`** function will return the locations of values that meet certain criteria.

```
match(c(25.85, 11.45, 9.23), soil.data$CEC)
```

```
## [1] 41 59 18
```

and to check the result...

```
soil.data[c(41, 59, 18), ]
```

```
##    PROFILE        Landclass Upper.Depth Lower.Depth clay silt sand
## 41       7         Cropping         0.7         0.8   47   11   42
## 59      10 improved pasture         0.1         0.2   47   12   42
## 18       3           Forest         0.7         0.8   37    9   54
##    pH_CaCl2 Total_Carbon    EC ESP ExchNa ExchK ExchCa ExchMg   CEC
## 41     6.70         0.23 0.063 2.2   0.61  0.53  14.22  12.95 25.85
## 59     5.94         0.70 0.039 0.1   0.01  0.74   6.76   2.61 11.45
## 18     6.05         0.17 0.088 0.7   0.06  0.33   6.15   2.35  9.23
```

Note that the **`match`** function matches the first observation only (this makes it difficult to use when there are multiple observations of the same value). This function is vectorised. The **`match`** function is useful for finding the location of the unique values, such as the maximum.

```
match(max(soil.data$CEC, na.rm = TRUE), soil.data$CEC)
```

```
## [1] 95
```

Note the call to the `na.rm` argument in the `max` function as a means to overlook the presence of `NA` values. So what is the maximum CEC value in our **`soil.data`** set.

```
soil.data$CEC[95]
```

```
## [1] 28.21
```

The **`which`** function, on the other hand, will return all locations that meet the criteria.

```
which(soil.data$ESP > 5)
```

```
##  [1]  68 101 102 103 104 107 108 109 111 113 114 117 118 123 146 147 148
## [18] 159
```

Of course, you can specify multiple constraints.

```
which(soil.data$ESP > 5 & soil.data$clay > 30)
```

```
## [1] 111 117 147 148 159
```

The `which` function can also be useful for locating missing values.

```
which(is.na(soil.data$ESP))
```

```
##  [1]   9  10  15  21  24  45  49  58  61  63  80  81  89 110 112 115 121
## [18] 129 138 150
```

```
soil.data$ESP[c(which(is.na(soil.data$ESP)))]
```

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

1.3 Factors

For many analyses, it is important to distinguish between quantitative (i.e. continuous) and categorical (i.e. discrete) variables. Categorical data are called factors in R. Internally, factors are stored as numeric data (as a check with `mode` will tell you), but they are handled as categorical data in statistical analyses. Factors are a class of data in R. R automatically recognizes non-numerical data as factors when the data are read in, but if numerical data are to be used as a factor (or if character data are generated within R and not read in, conversion to a factor must be specified explicitly. In R, the function `factor` does this.

```
a <- c(rep(0, 4), rep(1, 4))
a
```

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

```
a <- factor(a)
a
```

```
## [1] 0 0 0 0 1 1 1 1
## Levels: 0 1
```

The levels that R assigns to your factor are by default the unique values given in your original vector. This is often fine, but you may want to assign more meaningful levels. For example, say you have a vector that contains soil drainage class categories.

```
soil.drainage <- c("well drained", "imperfectly drained", "poorly drained",
    "poorly drained", "well drained", "poorly drained")
```

If you designate this as a factor, the default levels will be sorted alphabetically.

```
soil.drainage1 <- factor(soil.drainage)
soil.drainage1
```

```
## [1] well drained        imperfectly drained poorly drained
## [4] poorly drained      well drained        poorly drained
## Levels: imperfectly drained poorly drained well drained
```

```
as.numeric(soil.drainage1)
```

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

If you specify `levels` as an argument of the `factor` function, you can control the order of the levels.

```
soil.drainage2 <- factor(soil.drainage, levels = c("well drained",
"imperfectly drained", "poorly drained"))
as.numeric(soil.drainage2)
```

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

This can be useful for obtaining a logical order in statistical output or summaries.

1.4 Combining data

Data frames (or vectors or matrices) often need to be combined for analysis or plotting. Three `R` functions that are very useful for combining data are `rbind` and `cbind`. The function `rbind` simply "stacks" objects on top of each other to make a new object ("row bind"). The function `cbind` ("column bind") carries out an analogous operation with columns of data.

```
soil.info1 <- data.frame(soil = c("Vertosol", "Hydrosol", "Sodosol"), response = 1:3)
soil.info1
```

```
##       soil response
## 1 Vertosol        1
## 2 Hydrosol        2
## 3  Sodosol        3
```

```
soil.info2 <- data.frame(soil = c("Chromosol", "Dermosol", "Tenosol"), response = 4:6)
soil.info2
```

```
##        soil response
## 1 Chromosol        4
## 2  Dermosol        5
## 3   Tenosol        6
```

```
soil.info <- rbind(soil.info1, soil.info2)
soil.info
```

```
##        soil response
## 1  Vertosol        1
## 2  Hydrosol        2
## 3   Sodosol        3
```

```
## 4 Chromosol        4
## 5  Dermosol        5
## 6   Tenosol        6
```

```
a.column <- c(2.5, 3.2, 1.2, 2.1, 2, 0.5)
soil.info3 <- cbind(soil.info, SOC = a.column)
soil.info3
```

```
##        soil response SOC
## 1  Vertosol        1 2.5
## 2  Hydrosol        2 3.2
## 3   Sodosol        3 1.2
## 4 Chromosol        4 2.1
## 5  Dermosol        5 2.0
## 6   Tenosol        6 0.5
```

1.5 Exercises

1. Using the `soil.data` set, return the following:
   (a) The first 10 rows of the columns `clay`, `Total_Carbon`, and `ExchK`
   (b) The column `CEC` for the `Forest` land class
   (c) Use the `subset` function to create a new data frame that has only data for the cropping land class
   (d) How many soil profiles are there? Actually write some script to determine this rather than look at the data frame
2. Using the same data set, find the location and value of the maximum, minimum and median soil carbon (`Total_Carbon`)
3. Make a new data frame which is sorted by the upper soil depth (`Upper.Depth`). Can you sort it in decreasing order (Hint: Check the help file)
4. Make a new data frame which contains the columns `PROFILE`, `Landclass`, `Upper.Depth`, and `Lower.Depth`. Make another data frame which contains just the information regarding the exchangeable cations e.g. `ExchNa, ExchK, ExchCa`, and `ExchMg`. Make a new data frame which combines these two separate data frames together
5. Make separate data frame for each of the land classes. Now make a new data frame which combines the four separate data frames together.