Lab2 Instruction

September 30, 2015

Lab2 contains basics for file and directory manipulation, and R data input and output. Rather than a complete collection of functions, only these frequently used functions are mentioned in this instruction.

Note

- All R codes are present in boxes with grey background, in which lines leading by `#` will be treated as comments by R. You can run them in your R Console. Lines leading by `##` are outputs of R codes.
- The functions in **bold** are recommended to use in your data analysis.
- The Lab Instruction in different formats including .Rmd, .md, .html and .pdf are available on https://github.com/caijun/DAE/tree/master/Lab2. For your convenience, an R script named *Lab2Instruction.R* that contains all R codes occurred in the Lab is provided. The data used and results generated in the Lab are also provided. You can also download data from the course website. It is a zip file named *Lab2data*.

Part I File and Directory Manipulations

R has a variety of functions for file and directory manipulations. The following are a few examples:

`setwd()` and `getwd()`: used to change or determine the current working directory. It's a good habit to set working directory before your data analysis as all results generated in your analysis will be stored in the working directory.

`list.files()` and `list.dirs()`: returns a character vector of names of files or directories under the given directory.

`file.info()`: gives file size, creation time, directory vs. ordinary file status, and so on for each file whose name is in the argument, a character vector.

`file.create()` and `dir.create()`: creates files or directories with the given names if they do not already exist.

`file.exists()` and `dir.exists()`: returns a logical vector indicating whether the given file exists for each name in the first argument, a character vector.

`file.copy()` and `file.rename()`: moves files `from` source path `to` destination path.

`file.remove()` and **`unlink()`**: deletes the files or directories specified by the first argument, a character vector.

```
# set your working directory to Lab2
setwd("your/path/to/Lab2")
```

If you're working on Windows, you can set your working directory like `E:/R/DAE/Lab2`; or you're a Unix-like (such as Linux, Mac) OS user, your working directory can be something like `~/Documents/R/DAE/Lab2`. `getwd()` returns current working directory and you can use it to check whether your working directory is correctly set.

```r
getwd()
```

```
## [1] "/Users/tonytsai/Documents/R/DAE/Lab2"
```

Now you can use the aforementioned functions to manipulate files and directories under `Lab2`. Let's first use `list.files()` to see what `Lab2` contains.

```r
# list all files including directories under current working directory
list.files()
```

```
## [1] "data"                 "header.tex"           "Lab2Instruction.html"
## [4] "Lab2Instruction.md"   "Lab2Instruction.pdf"  "Lab2Instruction.Rmd"
## [7] "script"
```

`list.files()` lists all files including directories under `Lab2`. Assume that you're only interested in directories, you have to determine which of those are directories. `file.info()` can give you the file status that you need.

```r
# extract file information for those files
file.info(list.files())
```

```
##                        size isdir mode               mtime
## data                    374  TRUE  755 2015-09-21 01:09:20
## header.tex              317 FALSE  644 2015-09-22 17:20:20
## Lab2Instruction.html 349641 FALSE  644 2015-09-28 18:10:08
## Lab2Instruction.md    27847 FALSE  644 2015-09-28 18:10:08
## Lab2Instruction.pdf   79267 FALSE  644 2015-09-30 11:19:55
## Lab2Instruction.Rmd   22429 FALSE  644 2015-09-30 11:21:52
## script                  102  TRUE  755 2015-09-30 11:19:44
##                                    ctime               atime uid gid
## data                 2015-09-21 01:09:20 2015-09-30 11:19:44 501  20
## header.tex           2015-09-22 17:20:20 2015-09-30 11:19:46 501  20
## Lab2Instruction.html 2015-09-28 18:10:08 2015-09-28 22:06:19 501  20
## Lab2Instruction.md   2015-09-28 18:10:08 2015-09-28 22:06:19 501  20
## Lab2Instruction.pdf  2015-09-30 11:19:55 2015-09-30 11:19:55 501  20
## Lab2Instruction.Rmd  2015-09-30 11:21:52 2015-09-30 11:21:52 501  20
## script               2015-09-30 11:19:44 2015-09-30 11:19:44 501  20
##                         uname grname
## data                 tonytsai  staff
## header.tex           tonytsai  staff
## Lab2Instruction.html tonytsai  staff
## Lab2Instruction.md   tonytsai  staff
## Lab2Instruction.pdf  tonytsai  staff
## Lab2Instruction.Rmd  tonytsai  staff
## script               tonytsai  staff
```

The character vector of the names of files and directories returned by `list.files()` is used as the first argument of `file.info()`. Among the output file status, `isdir` indicates whether the file is a directory. You can use `list.dirs()` to filter only those directories.

```
# list only directories under current working directory
list.dirs()
```

```
## [1] "."              "./data"        "./data/CMDSSS" "./script"
```

`.` represents the current directory `Lab2`. Since the argument `recursive` in `list.dirs()` is defaultly set `TRUE`, you can see that `CMDSSS`, the directory under subdirectory `data`, is also listed out.

Another usage of `list.files()` is to find specific files under a given directory. For example, we want to find all R scripts under Lab2.

```
# find all R scripts under Lab2 and give their full path names (or absolute paths)
list.files(recursive = TRUE, pattern = ".R$", full.names = TRUE)
```

```
## [1] "./script/Lab2Instruction.R"
```

The argument `recursive = TRUE` means to search R scripts not only in current directory, but also in subdirectories. The argument `pattern` is usually a regular expression, which means only file names that match the regular expression will be returned. Here `pattern = ".R$"` filters those files that end with `.R"`. The argument `full.names = TRUE` returns the full path that the directory path is prepended to the file name. Turn `full.names` to `FALSE` and only the file name is returned.

During data analysis, it's common to create directories to write results. The following R code creates a recursive directory under Lab2 to store the TXT data, which will be used in Part III.

```
if(!dir.exists("data/CMDSSS"))
  dir.create("data/CMDSSS", recursive = TRUE)
```

We first use `dir.exists()` to check whether the directories to be created is existed. Make sure that they are not existed, we use `dir.create()` to create the recursive directories, otherwise there is no need to do such a directory creation.

Next we would like to create a temporary R script udner `script` to say hello to R. To accomplish such a task, we firstly create a temporary subdirectory named `tmp` under directory `script`, in which we further create a temporary R script named `tmp.R`.

```
# create a temporary directory under script
if(!dir.exists("script/tmp")) dir.create("script/tmp")
# create a temporary R script under tmp to say Hello World, Hello R!
file.create("script/tmp/tmp.R")
```

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

The `TRUE` value returned by `file.create()` indicates that the file `tmp.R` is successfully created.

Then, we can use `cat()` to ouput the R command `print('Hello World, Hello R!')`, which will print out "Hello World, Hello R!" on the R Console, to the file `tmp.R`.

```
cat("print('Hello World, Hello R!')", file = "script/tmp/tmp.R")
```

`source()` can read R code from a file or a connection and evaluate the parsed expressions sequentially. To excute the R code in `tmp.R`, type the following:

```r
source("script/tmp/tmp.R")
```

```
## [1] "Hello World, Hello R!"
```

```r
# copy tmp.R as helloworld.R in a differnt place
file.copy("script/tmp/tmp.R", "script/helloworld.R")
```

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

```r
list.files("script", recursive = TRUE)
```

```
## [1] "helloworld.R"      "Lab2Instruction.R" "tmp/tmp.R"
```

```r
# rename helloworld.R to hello.R
file.rename("script/helloworld.R", "script/hello.R")
```

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

```r
list.files("script", recursive = TRUE)
```

```
## [1] "hello.R"           "Lab2Instruction.R" "tmp/tmp.R"
```

```r
# delete all R scripts under script directory except for Lab2Instructon.R
# attempt to delete inexistent helloworld.R
file.remove(c("script/hello.R", "script/helloworld.R"))
```

```
## Warning in file.remove(c("script/hello.R", "script/helloworld.R")): cannot
## remove file 'script/helloworld.R', reason 'No such file or directory'
```

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

Here a character of two file paths is given to `file.remove()` and a logical vector of two elements is returned. As the second file path points to the inexistent `helloworld.R`, the corresponding `FALSE` indicates that the file deletion is failed. And a warning of “No such file or directory” is thrown to show the reason why failed to delete the file.

The following R codes show how to delete a directoty that is not empty. We can see that the `tmp` directory contains `tmp.R`. To delete such a directory, we can use `unlink()` with the argument `recursive` to be `TRUE`.

```r
list.files("script", recursive = TRUE)
```

```
## [1] "Lab2Instruction.R" "tmp/tmp.R"
```

```r
# delete the temporary directory that is not empty.
unlink("script/tmp", recursive = TRUE)
list.files("script")
```

```
## [1] "Lab2Instruction.R"
```

To see all the file- and directory-related functions, type the following:

```
?files
```

Part II Capturing R Console Output

R provides functions to save the results that appear in your R Console into a text file.

`sink()`: diverts R output to a file connection.

`capture.output()`: sends R output to a character string or file connection.

R also provides functions to save and load R objects in its own file formats, such as `.RData` or `.rda`.

`data()`: loads specified data sets, or list the available data sets.

`save()` and **`load()`**: writes R objects to the specified file. The objects can be read back from the file later by using `load()`. To save all the objects that you have created in current workspace, you can use `save.image()`, which is a wrapper function for `save(list = ls(all.names = TRUE), file = ".RData", envir = .GlobalEnv)`. The argument `envir` specifies in which environment R objects are to be saved; and `.GlobalEnv` represents the global environment, which is more often known as the user's workspace. It's common to see saving your workspace image before quitting R. If you choose "yes", then `save.image()` will write all R objects in your workspace into a file named `.RData` in your current working directory. The next time you start R in this directory, that workspace and all its data objects will be restored. You can easily continue your data analysis.

Create a new R script and type following codes. You're encouraged to write R codes in R script rather R Console as you can accumulate your R codes and easily modify them. Usually, you can type handy R functions and short R codes in R Console for interactive check.

```r
# divert R output to CO2.txt under data
sink(file = "data/CO2.txt")
# load R built-in dataset CO2. type ?CO2 to see the decription about CO2 dataset.
data("CO2")
# divert the first 14 rows of CO2
head(CO2, 14)
# end the divertion
sink()
```

Source the R script after saving it. `head(CO2, 14)` returns the first 14 rows of data.frame `CO2`, while you see the outputs are not displayed on R Console as the code is between `sink()`. Instead, the outputs are diverted into the file *CO2.txt*. It's always good to sink the outputs of your R codes into a file, particularly when it's time-consuming to run the R script once.

The following R command generates a vector containing a numeric sequence from 1 to 10.

```r
1:10
```

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

The R Console prints out 10 integers. We can use `capture.output()` to capture the output and asign it to a variable `x`.

```r
# send the output to variable x
x <- capture.output(1:10)
x
```

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

Printing out `x`, we can see variable `x` is a character string containing all the outputs returned by evaluating `1:10` including `[1]`.

Using `capture.output()` to divert the first 14 rows of data.frame `CO2` to *CO2.txt* is much easily done.

```
capture.output(head(CO2, 14), file = "data/CO2.txt")
```

Part III Imports and Exports

Here I list some typical functions in R for reading and storing data in common formats, such as Plain Text (.txt), CSV (.csv), Excel (.xls, .xlsx), and dBase (.dbf). For large and complex data, databases are supposed to be used for management.

`read.table()` and **`write.table()`**: reads a file in table format and creates a data frame from it.

`read.csv()` and **`write.csv()`**: reads a CSV file and creates a data frame from it. A standard CSV file actually stores tabular data in plain text with values separated by comma.

`read.xlsx()` and `write.xlsx()`: reads and writes Excel 2007 and Excel 97/2000/XP/2003 files. They are from xlsx package. There are some other packages for reading or/and Excel files, such as openxlsx, XLConnect, readxl and WriteXLS. Excel files do not work smoothly in many data processing software so CSV is a preferred format to save your data in MS Excel.

`read.dbf()` and `write.dbf()`: reads and writes dBase files. They are from foreign package, which provides functions for reading and writing data stored by Minitab, S, SAS, SPSS, Stata, dBase, …. The reason why I specially introduce how to read dBase files is that the attribute table of shape file is stored in .dbf format.

Now we use `read.table()` to read in *CO2.txt* to check the outputs of `sink()` in Part II.

```
# read data in Plain Text
txt <- read.table(file = "data/CO2.txt")
```

`str()` is a handy function for compactly displaying the structure of an arbitrary R object. Before manipulating an R object, it's always good to use `str` to show its structure, especially for thoses R objects that you aren't familar with.

```
str(txt)
```

```
## 'data.frame':    14 obs. of  5 variables:
##  $ Plant    : Factor w/ 2 levels "Qn1","Qn2": 1 1 1 1 1 1 1 2 2 2 ...
##  $ Type     : Factor w/ 1 level "Quebec": 1 1 1 1 1 1 1 1 1 1 ...
##  $ Treatment: Factor w/ 1 level "nonchilled": 1 1 1 1 1 1 1 1 1 1 ...
##  $ conc     : int  95 175 250 350 500 675 1000 95 175 250 ...
##  $ uptake   : num  16 30.4 34.8 37.2 35.3 39.2 39.7 13.6 27.3 37.1 ...
```

We can see `txt` is a data.frame with 14 observations of 5 variables. In a data.frame, a row represents an observation and a column represetn a variable. Hence, the data.frame `txt` has a size of 14 rows by 5 columns. The class of each variable and their values are also shown. Among which, the variable `Plant` is a factor with two value levels of `Qn1` and `Qn2`. Suppose that only `Qn1` plant is of interest. We can use `subset()` to extract data of `Qn1` plant.

```
Qn1 <- subset(txt, Plant == "Qn1")
```

The first argument `txt` is the object to be subsetted, and the second argument `Plant == "Qn1"` is a logical expression indicating that rows with `Plant` equal to `Qn1` are kept.

```
# save Qn1 to .txt.
# Though the file extension is .txt, the data is actually stored in a CSV format.
write.table(Qn1, file = "data/CO2-Qn1.txt", row.names = FALSE, quote = FALSE, sep = ",")
```

The argument `row.names = FALSE` indicates the row names of `Qn1` aren't to be written. The argument `quote = FALSE` represents the character or factor columns will not be surrounded by double quotes. It's necessary to turn `quote` to `TRUE` when the character column self contains the field separater. For example, the character variable `address` usually contains the field separater of CSV `,`. The argument `sep` specifies the field separator string, such as `,` and `\t`.

```
# save Qn1 to .csv
write.csv(Qn1, file = "data/CO2-Qn1.csv", row.names = FALSE, quote = FALSE)
# use read.csv to read CO2-Qn1.csv
csv <- read.csv(file = "data/CO2-Qn1.csv", header = TRUE, sep = ",",
                fileEncoding = "utf-8", stringsAsFactors = FALSE)
```

The argument `header` indicates whether the file contains the names of the variables as its first line. The argument `fileEncoding` declares the encoding used on the file, which must be careful when you read data containing Chinese on Windows. In such a situation, you can try `fileEncoding = "cp936"` or `fileEncoding = "GBK"`. It's always good to write and read files using `fileEncoding = "utf-8"`. `stringsAsFactors` is a logical value indicating whether character vectors should be converted to factors. The default is TRUE, while some functions return weird results when factors rather than characters are input.

```
str(csv)
```

```
## 'data.frame':    7 obs. of  5 variables:
##  $ Plant    : chr  "Qn1" "Qn1" "Qn1" "Qn1" ...
##  $ Type     : chr  "Quebec" "Quebec" "Quebec" "Quebec" ...
##  $ Treatment: chr  "nonchilled" "nonchilled" "nonchilled" "nonchilled" ...
##  $ conc     : int  95 175 250 350 500 675 1000
##  $ uptake   : num  16 30.4 34.8 37.2 35.3 39.2 39.7
```

You can see that the classes of variables `Plant`, `Type` and `Treatment` are all character rather than factor by turning `stringsAsFactors` to `FALSE`.

```
# install xlsx package
install.packages("xlsx")
```

```
# load xlsx pacakage
library(xlsx)
```

```
## Loading required package: rJava
## Loading required package: xlsxjars
```

```r
# write Qn1 to .xls
write.xlsx(Qn1, file = "data/CO2-Qn1.xls", sheetName = "Sheet1", col.names = TRUE,
           row.names = FALSE)
```

The argument `sheetName` is the sheet name of the workbook to be written. The argument `col.names` indicates if the column names of the data.frame to be written. We're used to writing column names rather than row names as the column names are the variable names.

```r
# use read.xlsx to read CO2-Qn1.xls
xlsx <- read.xlsx(file = "data/CO2-Qn1.xls", sheetName = "Sheet1")
str(xlsx)
```

```
## 'data.frame':    7 obs. of  5 variables:
##  $ Plant    : Factor w/ 1 level "Qn1": 1 1 1 1 1 1 1
##  $ Type     : Factor w/ 1 level "Quebec": 1 1 1 1 1 1 1
##  $ Treatment: Factor w/ 1 level "nonchilled": 1 1 1 1 1 1 1
##  $ conc     : num  95 175 250 350 500 675 1000
##  $ uptake   : num  16 30.4 34.8 37.2 35.3 39.2 39.7
```

```r
# install foreign package
install.packages("foreign")
```

```r
# load foreign pacakage
library(foreign)
# write Qn1 to .dbf
write.dbf(Qn1, file = "data/CO2-Qn1.dbf")
# use read.dbf to read CO2-Qn1.dbf
dbf <- read.dbf(file = "data/CO2-Qn1.dbf")
str(dbf)
```

```
## 'data.frame':    7 obs. of  5 variables:
##  $ Plant    : Factor w/ 1 level "Qn1": 1 1 1 1 1 1 1
##  $ Type     : Factor w/ 1 level "Quebec": 1 1 1 1 1 1 1
##  $ Treatment: Factor w/ 1 level "nonchilled": 1 1 1 1 1 1 1
##  $ conc     : int  95 175 250 350 500 675 1000
##  $ uptake   : num  16 30.4 34.8 37.2 35.3 39.2 39.7
##  - attr(*, "data_types")= chr  "C" "C" "C" "N" ...
```

`ls()` is also a handy function for listing objects in the specified environment.

```r
# list objects in current environment
ls()
```

```
## [1] "CO2"  "csv"  "dbf"  "Qn1"  "txt"  "x"    "xlsx"
```

```r
# save txt, csv, xlsx and dbf .RData or .rda
save(txt, csv, xlsx, dbf, file = "data/format.RData")
```

We use `rm()` to remove `txt`, `csv`, `xlsx` and `dbf` from current environment.

```r
rm(txt, csv, xlsx, dbf)
ls()
```

```
## [1] "CO2" "Qn1" "x"
```

Now we can load *format.RData* back and see all R objects in current environment.

```r
load(file = "data/format.RData")
ls()
```

Demostration

The `SURF_CLI_CHN_MUL_DAY-TEM-12001-201409.TXT` contains daily temporature collected from all meteorological stations across China in September 2014, which is downloaded from the dataset of **SURF_CLI_CHN_MUL_DAY** v3.0 produced by CMDSSS. The following code demostrates how to extract your desired data from **SURF_CLI_CHN_MUL_DAY** dataset in R, which is a common case that you may encounter in your own research.

```r
# use read.table to read dataset in TXT
data <- read.table(file = "data/CMDSSS/SURF_CLI_CHN_MUL_DAY-TEM-12001-201409.TXT")
str(data)
```

```
## 'data.frame':    24240 obs. of  13 variables:
##  $ V1 : int  50136 50136 50136 50136 50136 50136 50136 50136 50136 50136 ...
##  $ V2 : int  5258 5258 5258 5258 5258 5258 5258 5258 5258 5258 ...
##  $ V3 : int  12231 12231 12231 12231 12231 12231 12231 12231 12231 12231 ...
##  $ V4 : int  4385 4385 4385 4385 4385 4385 4385 4385 4385 4385 ...
##  $ V5 : int  2014 2014 2014 2014 2014 2014 2014 2014 2014 2014 ...
##  $ V6 : int  9 9 9 9 9 9 9 9 9 9 ...
##  $ V7 : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ V8 : int  94 83 82 153 152 157 170 144 89 80 ...
##  $ V9 : int  196 219 144 245 264 242 296 219 166 227 ...
##  $ V10: int  29 1 4 97 85 67 57 104 47 -33 ...
##  $ V11: int  0 0 0 0 0 0 0 0 0 0 ...
##  $ V12: int  0 0 0 0 0 0 0 0 0 0 ...
##  $ V13: int  0 0 0 0 0 0 0 0 0 0 ...
```

```r
# V1: station id
# V2: latitude, the last two digits are minitue and the remaining digits are degrees
# V3: longitude, the last two digits are minitue and the remaining digits are degrees
# V4: altitude in 0.1 meter. When the station altitude is estimated rather than
#     measured, 100000 is added.
# V5: year
# V6: month
# V7: days of the month
# V8: average temperature in 0.1 degree Celsius
# V9: daily maximum temperature in 0.1 degree Celsius
# V10: daily minimum temperature in 0.1 degress Celsius.
#      When the actual temperature is higher than the higher limit of the instrument
#      measuring range, 10000 is added; When the actual temperature is lower than the
#      lower limit, 10000 is subtracted.
```

```
#      32766 indicates observations are not available (NA).
# V11: quality control code of average temperature
# V12: quality control code of daily maximum temperature
# V13: quality control code of daily minimum temperature
```

As `SURF_CLI_CHN_MUL_DAY-TEM-12001-201409.TXT` doesn't contain a header, the column names of `V1, V2, ..., V13` are automatically added. When no header is contained in the file to be read, column names comprised of `V` and column index are provided. The property of a good variable name is that you're able to know its meanning from the name. Apparently, `V1, V2, ..., V13` are not good variable names. We can use `colnames()` to set and retrieve column names of a data.frame.

```
colnames(data) <- c("id", "lat", "lng", "alt", "year", "month", "day", "meanT",
                    "maxT", "minT", "meanQC", "maxQC", "minQC")
colnames(data)
```

```
##  [1] "id"     "lat"    "lng"    "alt"    "year"   "month"  "day"
##  [8] "meanT"  "maxT"   "minT"   "meanQC" "maxQC"  "minQC"
```

Suppose that we are interested in temperatures from Miyun station whose id is 54416. Here we use `[` operator to extract our desired data. `[` is an operator acting on vectors, matrices, arrays and lists to extract or replace parts.

We know `data` is a data.frame with 24240 observations of 13 variables from `str(data)`. On a technical level, a data.frame is a list, with the components of that list being equal-length vectors. Hence, we can access a data.frame via component index values or component names. For example, `data[[1]]` represents extracting the first variable, namely `id` from `data` via component index `1`; `data$id` will extract the variable `id` by component name `id`. It's notable that `[[` is supposed to be used to extract the integer vector of variable `id`, otherwise only using `[` will return a subsetting data.frame with only one variable `id`.

```
str(data[[1]])
```

```
##  int [1:24240] 50136 50136 50136 50136 50136 50136 50136 50136 50136 50136 ...
```

```
str(data[1])
```

```
## 'data.frame':    24240 obs. of  1 variable:
##  $ id: int  50136 50136 50136 50136 50136 50136 50136 50136 50136 50136 ...
```

We can also treat data.frame in a matrix-like fashion. For example, we can extract the 8th to 10th columns.

```
str(data[, 8:10])
```

```
## 'data.frame':    24240 obs. of  3 variables:
##  $ meanT: int  94 83 82 153 152 157 170 144 89 80 ...
##  $ maxT : int  196 219 144 245 264 242 296 219 166 227 ...
##  $ minT : int  29 1 4 97 85 67 57 104 47 -33 ...
```

The first subscript being empty means all rows will be kept. Now we have all temperature variables. We can also extract those temperature variables by giving their column names.

```r
str(data[, c("meanT", "maxT", "minT")])
```

```
## 'data.frame':    24240 obs. of  3 variables:
##  $ meanT: int  94 83 82 153 152 157 170 144 89 80 ...
##  $ maxT : int  196 219 144 245 264 242 296 219 166 227 ...
##  $ minT : int  29 1 4 97 85 67 57 104 47 -33 ...
```

But our goal is to acquire temperatures collected from Miyun station. Therefore, we have to further keep only those rows with variable `id` equal to `54416`.

```r
# extract temperatures collected from Miyun station whose id is 54416
Miyun.TEM <- data[data$id == 54416, c("meanT", "maxT", "minT")]
```

`data$id == 54416` returns a logical vector indicating whether the variable `id` in the observation is equal to 54416. Taking a logical vector into the first subsript means only rows with `TRUE` value will be kept.

```r
str(Miyun.TEM)
```

```
## 'data.frame':    30 obs. of  3 variables:
##  $ meanT: int  222 195 206 225 229 235 232 215 204 208 ...
##  $ maxT : int  259 225 294 329 292 287 281 291 289 282 ...
##  $ minT : int  201 186 127 151 174 191 204 123 133 156 ...
```

We can summarize the extracted data by `summary()`, which gives us the statistics of minimum, 1st quantile, mean, median, 3rd quantile, and maximum of the input numeric vector.

```r
# summaries of meanT, maxT, and minT
summary(Miyun.TEM)
```

```
##      meanT            maxT            minT      
##  Min.   :102.0   Min.   :146.0   Min.   : 68.0  
##  1st Qu.:171.0   1st Qu.:225.8   1st Qu.:122.2  
##  Median :194.5   Median :252.5   Median :148.5  
##  Mean   :190.4   Mean   :250.4   Mean   :145.2  
##  3rd Qu.:206.0   3rd Qu.:280.5   3rd Qu.:167.8  
##  Max.   :235.0   Max.   :329.0   Max.   :204.0
```

Oops, you see that the temperatue values are too large as they are in 0.1 degree Celsius. Multiplying all temperatures by a scale of 0.1 gives us the actual temperature values. Since R supports vectorized arithmetic operations, multiplying each element in the data.frame by a numeric can be easily accomplished without the typical `for` loops.

```r
Miyun.TEM <- Miyun.TEM * 0.1
summary(Miyun.TEM)
```

```
##      meanT            maxT            minT      
##  Min.   :10.20   Min.   :14.60   Min.   : 6.80  
##  1st Qu.:17.10   1st Qu.:22.57   1st Qu.:12.22  
##  Median :19.45   Median :25.25   Median :14.85  
##  Mean   :19.04   Mean   :25.04   Mean   :14.52  
##  3rd Qu.:20.60   3rd Qu.:28.05   3rd Qu.:16.77  
##  Max.   :23.50   Max.   :32.90   Max.   :20.40
```

Now we can save the extracted temperatures from Miyun station to external file for further data analysis.

```
save(Miyun.TEM, file = "data/Miyun-TEM-201409.RData")
```

References

The following are materials on R data import/export that you can access on the Web.

- R Data Import/Export