```
---
title: "Chapter 3: Data IO(Input/Output) & Subsetting"
author: "M Affouf"
date: "2/6/2018"
output:output: html_document
---

#part1


## Before we get Started: Working Directories

* R looks for files on your computer relative to the "working" directory
* It's always safer to set the working directory at the beginning of your script. Note that setting the working directory created the necessary code that you can copy into your script.
* Example of help file

```{r workingDirectory,eval=FALSE}
## get the working directory
getwd()
#
```


Setting a Working Directory

* Setting the directory can sometimes be finicky
    * Windows: Default directory structure involves single backslashes (" \\ "), but R interprets these as "escape" characters. So you must replace the backslash with forward slashed (" / ") or two backslashes (" \\ \\ ")
    * Mac/Linux: Default is forward slashes, so you are okay
* Typical linux/DOS directory structure syntax applies
    * ".." goes up one level
    * "./" is the current directory
    * "~" is your home directory

Working Directory

Note that the `dir()` function interfaces with your operating system and can show you which files are in your current working directory.

You can try some directory navigation:

```{r directoryNav}
dir("./") # shows directory contents
dir("..")
```


Working Directory

* Copy the code to set your working directory from the History tab in RStudio (top right)
* Confirm the directory contains "day1.R" using `dir()`

Data Input
```

* 'Reading in' data is the first step of any real project/analysis
* R can read almost any file format, especially via add-on packages
* We are going to focus on simple delimited files first
    * tab delimited (e.g. '.txt')
    * comma separated (e.g. '.csv')
    * Microsoft excel (e.g. '.xlsx')

Data Aside

* Everything we do in class will be using real publicly available data - there are few 'toy' example datasets and 'simulated' data
* OpenBaltimore and Data.gov will be sources in the first few chapters

Data Input

Monuments Dataset: "This data set shows the point location of Baltimore City monuments. However, the completness and currentness of these data are uncertain."

* Download data from https://data.baltimorecity.gov/Community/Monuments/cpxf-kxp3)

Data Input
R Studio features some nice "drop down" support, where you can run some tasks by selecting them from the toolbar.

For example, you can easily import text datasets using the "Tools --> Import Dataset" command. Selecting this will bring up a new screen that lets you specify the formatting of your text file.

After importing a datatset, you get the corresponding R commands that you can enter in the console if you want to re-import data.

Data Input {.smaller}

So what is going on "behind the scenes"?

`read.table()`: Reads a file in table format and creates a data frame from it, with cases corresponding to lines and variables to fields in the file.

```
# the four ones I've put at the top are the important inputs
read.table( file, # filename
           header = FALSE, # are there column names?
           sep = "", # what separates columns?
           as.is = !stringsAsFactors, # do you want character strings as factors or characters?
           quote = "\"'",  dec = ".", row.names, col.names,
           na.strings = "NA", nrows = -1,
           skip = 0, check.names = TRUE, fill = !blank.lines.skip,
           strip.white = FALSE, blank.lines.skip = TRUE, comment.char = "#",
           stringsAsFactors = default.stringsAsFactors())

# for example: `read.table("file.txt", header = TRUE, sep="\t", as.is=TRUE)`
```

```

## Data Input

* The filename is the path to your file, in quotes
* The function will look in your "working directory" if no absolute file path is given
* Note that the filename can also be a path to a file on a website (e.g. 'www.someurl.com/table1.txt')


## Data Input

There is a 'wrapper' function for reading CSV files:

```{r readCSV}
read.csv
```

Note: the `...` designates extra/optional arguments that can be passed to `read.table()` if needed

Data Input {.smaller}

* Here would be reading in the data from the command line, specifying the file path:

```{r readCSV2}
mon = read.csv("Monuments.csv",header=TRUE,as.is=TRUE)
head(mon)
```

Data Input {.smaller}

```{r subset5}
colnames(mon) # column names
head(mon$zipCode) # first few rows
```

Data Input

The `read.table()` function returns a `data.frame`, which is the primary data format for most data cleaning and analyses

```{r readCSV3}
str(mon) # structure of an R object
```

Data Input

Changing variable names in `data.frame`s works using the `names()` function, which is analagous to `colnames()` for data frames (they can be used interchangeably)

```{r names1}
names(mon)[1] = "Name"
names(mon)
```
```

```
names(mon)[1] = "name"
names(mon)
```

Data Output {.smaller}

While its nice to be able to read in a variety of data formats, it's equally important to be able to output data somewhere.

`write.table()`: prints its required argument `x` (after converting it to a `data.frame` if it is not one nor a `matrix`) to a file or connection.

```
write.table(x,file = "", append = FALSE, quote = TRUE, sep = " ",
            eol = "\n", na = "NA", dec = ".", row.names = TRUE,
            col.names = TRUE, qmethod = c("escape", "double"),
            fileEncoding = "")
```

Data Output

`x`: the R `data.frame` or `matrix` you want to write

`file`: the file name where you want to R object written. It can be an absolute path, or a filename (which writes the file to your working directory)

`sep`: what character separates the columns?

* "," = .csv - Note there is also a `write.csv()` function
* "\t" = tab delimited

`row.names`: I like setting this to FALSE because I email these to collaborators who open them in Excel

Data Output {.smaller}

For example, we can write back out the Monuments dataset with the new column name:

```{r writecsv}
names(mon)[6] = "Location"
write.csv(mon, file="monuments_newNames.csv", row.names=FALSE)
```

Note that `row.names=TRUE` would make the first column contain the row names, here just the numbers `1:nrow(mon)`, which is not very useful for Excel. Note that row names can be useful/informative in R if they contain information (but then they would just be a separate column).

Data Input - Excel

Many data analysts collaborate with researchers who use Excel to enter and curate their data. Often times, this is the input data for an analysis. You therefore have two options for getting this data into R:

* Saving the Excel sheet as a .csv file, and using `read.csv()`
* Using an add-on package, like `xlsx`, `readxl`, or `openxlsx`

For single worksheet .xlsx files, I often just save the spreadsheet as a .csv file (because I often have to strip off additional summary data from the columns)

For an .xlsx file with multiple well-formated worksheets, I use the `xlsx`, `readxl`, or `openxlsx` package for reading in the data.

Data Input - Other Software

* **haven** package (https://cran.r-project.org/web/packages/haven/index.html) reads in SAS, SPSS, Stata formats
* **readxl** package - the `read_excel` function can read Excel sheets easily
* **readr** package - Has *read_csv*/*write_csv* and *read_table* functions similar to *read.csv*/*write.csv* and *read.table*. Has different defaults, but can read **much faster** for very large data sets
* **sas7bdat** reads .sas7bdat files
* **foreign** package - can read all the formats as **haven**. Around longer (aka more testing), but not as maintained (bad for future).

#part2

Data Output {.smaller}

While its nice to be able to read in a variety of data formats, it's equally important to be able to output data somewhere.

`write.table()`: prints its required argument `x` (after converting it to a `data.frame` if it is not one nor a `matrix`) to a file or connection.

```
write.table(x,file = "", append = FALSE, quote = TRUE, sep = " ",
            eol = "\n", na = "NA", dec = ".", row.names = TRUE,
            col.names = TRUE, qmethod = c("escape", "double"),
            fileEncoding = "")
```

Data Output

`x`: the R `data.frame` or `matrix` you want to write

`file`: the file name where you want to R object written. It can be an absolute path, or a filename (which writes the file to your working directory)

`sep`: what character separates the columns?

* "," = .csv - Note there is also a `write.csv()` function

* "\t" = tab delimited

`row.names`: I like setting this to FALSE because I email these to collaborators who open them in Excel

Data Output {.smaller}

For example, from the Homework 2 Dataset:

```{r writecsv2}
circ = read.csv("Charm_City_Circulator_Ridership.csv",
header=TRUE,as.is=TRUE)
circ2 = circ[,c("day","date",
"orangeAverage","purpleAverage","greenAverage",
                "bannerAverage","daily")]
write.csv(circ2, file="charmcitycirc_reduced.csv", row.names=FALSE)
```

Note that `row.names=TRUE` would make the first column contain the row names, here just the numbers `1:nrow(circ2)`, which is not very useful for Excel. Note that row names can be useful/informative in R if they contain information (but then they would just be a separate column).

Data Input - Excel

Many data analysts collaborate with researchers who use Excel to enter and curate their data. Often times, this is the input data for an analysis. You therefore have two options for getting this data into R:

* Saving the Excel sheet as a .csv file, and using `read.csv()`
* Using an add-on package called `xlsx`

For single worksheet .xlsx files, I often just save the spreadsheet as a .csv file (because I often have to strip off additional summary data from the columns)

For an .xlsx file with multiple well-formated worksheets, I use the `xlsx` package for reading in the data.

More on Packages

Packages are add-ons that are commonly written by users comprised of functions, data, and vignettes

* Use `library()` or `require()` to load the package into memory so you can use its functions
* Install packages using `install.packages("PackageName")`
* Use `help(package="PackageName")` to see what contents the package has
* http://cran.r-project.org/web/packages/available_packages_by_name.html

More on Packages

Some useful data input/output packages

* foreign package - read data from Stata/SPSS/SAS
* sas7bdat - read SAS data

* xlsx - reads in XLS files

Installing Packages

```{r xlsx1,eval=FALSE}
install.packages("xlsx") # OR:
#install.packages("xlsx",
#    repos="http://cran.us.r-project.org")
#library(xlsx) # or require(xlsx)
```

Note you will need a stand-alone version of Java to use this

Saving R Data {.smaller}

It's very useful to be able to save collections of R objects for future analyses.

For example, if a task takes several hours(/days) to run, it might be nice to run it once and save the results for downstream analyses.

`save(...,file="[name].rda")`

where `...` is as many R objects, referenced by unquoted variable names, as you want to save.

For example, from the homework:

```{r save1}
save(circ,circ2,file="charmcirc.rda")
```

Saving R Data

You also probably have noticed the prompt when you close R about saving your workspace. The workspace is the collection of R objects and custom R functions in your current environment. You can check the workspace with `ls()` or view it in the "Workspace" tab:

```{r ls}
ls()
```

Saving R Data

Saving the workspace will save all of these files in your current working directory as a hidden file called ".Rdata". The function `save.image()` also saves the entire workspace, but you can give your desired file name as an input (which is nicer because the file is not hidden).

Note that R Studio should be able to open any .rda or .Rdata file. Opening one of these file types from Windows Explorer or OSX's Finder loads all of the objects into your workspace and changes your working directory to wherever the file was located.

Loading R Data

You can easily load any '.rda' or '.Rdata' file with the `load()` function:

```{r loadData}
tmp=load("charmcirc.rda")
tmp
ls()
```

Note that this saves the R object names as character strings in an object called 'tmp', which is nice if you already have a lot of items in your working directory, and/or you don't know exactly which got loaded in

Removing R Data

You can easily remove any R object(s) using the `rm()` or `remove()` functions, and they are no longer in your R environment (which you can confirm with running `ls()`)

You can also remove all of the objects you have added to your workplace with:

`> rm(list = ls())`

Subsetting Data

Often you only want to look at subsets of a data set at any given time. As a review, elements of an R object are selected using the brackets.

Today we are going to look at more flexible ways of identifying which rows of a dataset to select.

Subsetting Data

You can put a `-` before integers inside brackets to remove these indices from the data.

```{r }
x = c(1,3,77,54,23,7,76,5)
x[1:3] # first 3
x[-2] # all but the second
```

Subsetting Data

Note that you have to be careful with this syntax when dropping more than 1 element:

```{r }
x[-c(1,2,3)] # drop first 3
# x[-1:3] # shorthand. R sees as -1 to 3
x[-(1:3)] # needs parentheses
```

Selecting on multiple queries

What about selecting rows based on the values of two variables? We can 'chain' together logical statements using the following:

* `&` : AND
* `|` : OR

```{r andEx}
# which Mondays had more than 3000 average riders?
which(circ$day =="Monday" & circ$daily > 3000)[1:20]
```

AND {.smaller}

Which days had more than 10000 riders overall and more than 3000 riders on the purple line?

```{r andEx2}
Index=which(circ$daily > 10000 & circ$purpleAverage > 3000)
length(Index) # the number of days
head(circ[Index,],2) # first 2 rows
```

OR {.smaller}

Which days had more than 10000 riders overall or more than 3000 riders on the purple line?

```{r orEx1}
Index=which(circ$daily > 10000 | circ$purpleAverage > 3000)
length(Index) # the number of days
head(circ[Index,],2) # first 2 rows
```

Subsetting with missing data

Note that logical statements cannot evaluate missing values, and therefore returns an `NA`:

```{r naEval}
circ$purpleAverage[1:10] > 0
which(circ$purpleAverage > 0)[1:10]
```

Subsetting with missing data

You can use the `complete.cases()` function on a data frame, matrix, or vector, which returns a logical vector indicating which cases are complete, i.e., they have no missing values.

Selecting on multiple categories {.smaller}

You can select rows where a value is allowed to be several categories. In the homework, we had to subset the Charm City Circulator dataset by each day. How can we select rows that are 1 of 2 days?

The `%in%` operator proves useful: "`%in%` is a more intuitive interface as a binary operator, which returns a logical vector indicating if there is a match or not for its left operand." It also returns `FALSE` for `NAs`

```{r inEx}
(circ$day %in% c("Monday","Tuesday"))[1:20] # select entries that are
```

monday or tuesday
which(circ$day %in% c("Monday","Tuesday"))[1:20] # which indices are true?
```

## Subsetting columns

We touched on this last class. You can select columns using the variable/column names or column index

```{r colSelect}
circ[1:3, c("purpleAverage","orangeAverage")]
circ[1:3, c(7,5)]
```

Subsetting columns {.smaller}

You can also remove a column by setting its value to NULL

```{r colRemove}
tmp = circ2
tmp$daily=NULL
tmp[1:3,]
```

Select specific elements using an index

Often you only want to look at subsets of a data set at any given time. As a review, elements of an R object are selected using the brackets (`[` and `]`).

For example, `x` is a vector of numbers and we can select the second element of `x` using the brackets and an index (2):

```{r}
x = c(1, 4, 2, 8, 10)
x[2]
```

Select specific elements using an index

We can select the fifth or second AND fifth elements below:

```{r}
x = c(1, 2, 4, 8, 10)
x[5]
x[c(2,5)]
```

Subsetting by deletion of entries

You can put a minus (`-`) before integers inside brackets to remove these
```

indices from the data.

```{r negativeIndex}
x[-2] # all but the second
```
Note that you have to be careful with this syntax when dropping more than 1 element:

```{r negativeIndex2}
x[-c(1,2,3)] # drop first 3
# x[-1:3] # shorthand. R sees as -1 to 3
x[-(1:3)] # needs parentheses
```

Select specific elements using logical operators

What about selecting rows based on the values of two variables? We use logical statements. Here we select only elements of `x` greater than 2:

```{r}
x
x > 2
x[ x > 2 ]
```

Select specific elements using logical operators
You can have multiple logical conditions using the following:

* `&` : AND
* `|` : OR

```{r}
x[ x > 2 & x < 5 ]
x[ x > 5 | x == 2 ]
```

which function

The `which` functions takes in logical vectors and returns the index for the elements where the logical value is `TRUE`.

```{r}
which(x > 5 | x == 2) # returns index
x[ which(x > 5 | x == 2) ]
x[ x > 5 | x == 2 ]
```

Creating a `data.frame` to work with

Here we create a toy data.frame named `df` using random data:
```{r}
set.seed(2016) # reproducbility
df = data.frame(x = c(1, 2, 4, 10, 10),
```

```
                x2 = rpois(5, 10),
                y = rnorm(5),
                z = rpois(5, 6)
                )
```


Renaming Columns

Renaming Columns of a `data.frame`: base R

We can use the `colnames` function to directly reassign column names of
`df`:
```{r}
colnames(df) = c("x", "X", "y", "z")
head(df)
colnames(df) = c("x", "x2", "y", "z") #reset
```

Renaming Columns of a `data.frame`: base R

We can assign the column names, change the ones we want, and then re-
assign
the column names:
```{r}
cn = colnames(df)
cn[ cn == "x2"] = "X"
colnames(df) = cn
head(df)
colnames(df) = c("x", "x2", "y", "z") #reset
```

Renaming Columns of a `data.frame`: dplyr


```{r}
library(dplyr)
```

Note, when loading `dplyr`, it says objects can be "masked". That means
if you
use a function defined in 2 places, it uses the one that is loaded in
last.

Renaming Columns of a `data.frame`: dplyr

For example, if we print `filter`, then we see at the bottom
`namespace:dplyr`, which
means when you type `filter`, it will use the one from the `dplyr`
package.

```{r}
filter
```
Renaming Columns of a `data.frame`: dplyr

A `filter` function exists by default in the `stats` package, however. If

you want
to make sure you use that one, you use `PackageName::Function` with the
colon-colon
("`::`") operator.

```{r}
head(stats::filter,2)
```

This is important when loading many packages, and you may have
some conflicts/masking:

Renaming Columns of a `data.frame`: dplyr

To rename columns in `dplyr`, you use the `rename` command

```{r}
df = dplyr::rename(df, X = x2)
head(df)
df = dplyr::rename(df, x2 = X) # reset
```

Subsetting Columns

Subset columns of a `data.frame`:

We can grab the `x` column using the `$` operator.

```{r}
df$x
```

Subset columns of a `data.frame`:

We can also subset a `data.frame` using the bracket `[, ]` subsetting.

For `data.frame`s and matrices (2-dimensional objects), the brackets are
`[rows, columns]` subsetting. We can grab the `x` column using the index
of the column or the column name ("`x`")

```{r}
df[, 1]
df[, "x"]
```

Subset columns of a `data.frame`:

We can select multiple columns using multiple column names:

```{r}
df[, c("x", "y")]
```

Subset columns of a `data.frame`: dplyr

The `select` command from `dplyr` allows you to subset
```{r}
select(df, x)
```

Select columns of a `data.frame`: dplyr

The `select` command from `dplyr` allows you to subset columns of
```{r}
select(df, x, x2)
select(df, starts_with("x"))
```

Subsetting Rows

Subset rows of a `data.frame` with indices:

Let's select **rows** 1 and 3 from `df` using brackets:

```{r}
df[ c(1, 3), ]
```

Subset rows of a `data.frame`:

Let's select the rows of `df` where the `x` column is greater than 5 or is equal to 2. Without any index for columns, all columns are returned:

```{r}
df[ df$x > 5 | df$x == 2, ]
```

Subset rows of a `data.frame`:

We can subset both rows and colums at the same time:

```{r}
df[ df$x > 5 | df$x == 2, c("y", "z")]
```

Subset rows of a `data.frame`: dplyr

The command in `dplyr` for subsetting rows is `filter`. Try `?filter`

```{r}
filter(df, x > 5 | x == 2)
```
Note, no `$` or subsetting is necessary. R "knows" `x` refers to a column of `df`.

Subset rows of a `data.frame`: dplyr

By default, you can separate conditions by commas, and `filter` assumes

these statements are joined by `&`

```{r}
filter(df, x > 2 & y < 0)
filter(df, x > 2, y < 0)
```

Combining `filter` and `select`

You can combine `filter` and `select` to subset the rows and columns, respectively, of a `data.frame`:

```{r}
select(filter(df, x > 2 & y < 0), y, z)
```

In `R`, the common way to perform multiple operations is to wrap functions around each other in a nested way such as above

Assigning Temporary Objects

One can also create temporary objects and reassign them:

```{r}
df2 = filter(df, x > 2 & y < 0)
df2 = select(df2, y, z)
```

Piping - a new concept

There is another (newer) way of performing these operations, called "piping". It is becoming more popular as it's easier to read:

```{r}
df %>% filter(x > 2 & y < 0) %>% select(y, z)
```

It is read: "take df, then filter the rows and then select `y`, `z`".

Adding/Removing Columns

Adding new columns to a `data.frame`: base R

You can add a new column, called `newcol` to `df`, using the `$` operator:
```{r}
df$newcol = 5:1
df$newcol = df$x + 2
```

Removing columns to a `data.frame`: base R

You can remove a column by assigning to `NULL`:
```{r}
df$newcol = NULL
```

```
or selecing only the columns that were not `newcol`:
```{r}
df = df[, colnames(df) != "newcol"]
```

Adding new columns to a `data.frame`: base R

You can also "**c**olumn **bind**" a `data.frame` with a vector (or series of vectors), using the `cbind` command:
```{r}
cbind(df, newcol = 5:1)
```


Adding columns to a `data.frame`: dplyr

The `mutate` function in `dplyr` allows you to add or replace columns of a `data.frame`:
```{r}
mutate(df, newcol = 5:1)
print({df = mutate(df, newcol = x + 2)})
```

Removing columns to a `data.frame`: dplyr

The `NULL` method is still very common.

The `select` function can remove a column with a minus (`-`), much like removing rows:
```{r}
select(df, -newcol)
```

Removing columns to a `data.frame`: dplyr

Remove `newcol` and `y`
```{r}
select(df, -one_of("newcol", "y"))
```

Ordering columns

Ordering the columns of a `data.frame`: base R

We can use the `colnames` function to get the column names of `df` and then put `newcol` first by subsetting `df` using brackets:

```{r}
cn = colnames(df)
df[, c("newcol", cn[cn != "newcol"]) ]
```

Ordering the columns of a `data.frame`: dplyr

The `select` function can reorder columns. Put `newcol` first, then select the rest of columns:
```

```{r}
select(df, newcol, everything())
```

Ordering rows

Ordering the rows of a `data.frame`: base R

We use the `order` function on a vector or set of vectors, in increasing order:

```{r}
df[ order(df$x), ]
```

Ordering the rows of a `data.frame`: base R

The `decreasing` argument will order it in decreasing order:
```{r}
df[ order(df$x, decreasing = TRUE), ]
```

Ordering the rows of a `data.frame`: base R

You can pass multiple vectors, and must use the negative (using `-`) to mix decreasing and increasing orderings (sort increasing on `x` and decreasing on `y`):
```{r}
df[ order(df$x, -df$y), ]
```

Ordering the rows of a `data.frame`: dplyr

The `arrange` function can reorder rows By default, `arrange` orders in ascending order:
```{r}
arrange(df, x)
```

Ordering the rows of a `data.frame`: dplyr
Use the `desc` to arrange the rows in descending order:
```{r}
arrange(df, desc(x))
```

Ordering the rows of a `data.frame`: dplyr
It is a bit more straightforward to mix increasing and decreasing orderings:
```{r}
arrange(df, x, desc(y))
```

Transmutation

The `transmute` function in `dplyr` combines both the `mutate` and `select` functions. One can create new columns and keep the only the columns wanted:

```{r}
transmute(df, newcol2 = x * 3, x, y)
```