DATA ANALYTICS AND R PROGRAMMING

Subject code: 18MIT22C

UNIT-IV: Getting Started with R- R Nuts and Bolts - Getting Data in and Out of R - Using Textual and Binary Formats for Storing Data- Interfaces to the Outside World- Subsetting R Objects - Vectorized Operations - Managing Data Frames with the dplyr package.

Text book:

1. Roger D. Peng, “R Programming for Data Science” Lean Publishing, 2014. (Unit IV & V)

Prepared by Dr.P.Sumathi

R Nuts and Bolts

Entering Input

At the R prompt we type expressions. The <- symbol is the assignment operator.

```
> x <- 1
> print(x)
[1] 1
> x
[1] 1
> msg <- "hello"
```

The grammar of the language determines whether an expression is complete or not.

```
x <-    ## Incomplete expression
```

The # character indicates a comment. Anything to the right of the # (including the # itself) is ignored. This is the only comment character in R. Unlike some other languages, R does not support multi-line comments or comment blocks.

Evaluation

When a complete expression is entered at the prompt, it is evaluated and the result of the evaluated expression is returned. The result may be *auto-printed*.

```
> x <- 5      ## nothing printed
> x   ## auto-printing occurs
[1] 5
> print(x)      ## explicit printing
[1] 5
```

The [1] shown in the output indicates that x is a vector and 5 is its first element.

Typically with interactive work, we do not explicitly print objects with the print function; it is much easier to just auto-print them by typing the name of the object and hitting return/enter. However, when writing scripts, functions, or longer programs, there is sometimes a need to explicitly print objects because auto-printing does not work in those settings.

When an R vector is printed you will notice that an index for the vector is printed in square brackets [] on the side. For example, see this integer sequence of length 20.

```
> x <- 10:30
> x
 [1] 10 11 12 13 14 15 16 17 18 19 20 21
[13] 22 23 24 25 26 27 28 29 30
```

The numbers in the square brackets are not part of the vector itself, they are merely part of the *printed output*.

With R, it's important that one understand that there is a difference between the actual R object and the manner in which that R object is printed to the console. Often, the printed output may have additional bells and whistles to make the output more friendly to the users. However, these bells and whistles are not inherently part of the object.

Note that the : operator is used to create integer sequences.

R Objects

R has five basic or "atomic" classes of objects:

- character
- numeric (real numbers)
- integer
- complex
- logical (True/False)

The most basic type of R object is a vector. Empty vectors can be created with the vector() function. There is really only one rule about vectors in R, which is that **A vector can only contain objects of the same class**.

But of course, like any good rule, there is an exception, which is a *list*, which we will get to a bit later. A list is represented as a vector but can contain objects of different classes. Indeed, that's usually why we use them.

There is also a class for "raw" objects, but they are not commonly used directly in data analysis and I won't cover them here.

Numbers

Numbers in R are generally treated as numeric objects (i.e. double precision real numbers). This means that even if you see a number like "1" or "2" in R, which you might think of as integers, they are likely represented behind the scenes as numeric objects (so something like "1.00" or "2.00"). This isn't important most of the time…except when it is.

If you explicitly want an integer, you need to specify the L suffix. So entering 1 in R gives you a numeric object; entering 1L explicitly gives you an integer object.

There is also a special number `Inf` which represents infinity. This allows us to represent entities like `1 / 0`. This way, `Inf` can be used in ordinary calculations; e.g. `1 / Inf` is 0.

The value NaN represents an undefined value ("not a number"); e.g. 0 / 0; NaN can also be thought of as a missing value (more on that later)

Attributes

R objects can have attributes, which are like metadata for the object. These metadata can be very useful in that they help to describe the object. For example, column names on a data frame help to tell us what data are contained in each of the columns. Some examples of R object attributes are

- names, dimnames
- dimensions (e.g. matrices, arrays)
- class (e.g. integer, numeric)
- length
- other user-defined attributes/metadata

Attributes of an object (if any) can be accessed using the `attributes()` function. Not all R objects contain attributes, in which case the `attributes()` function returns NULL.

Creating Vectors

The `c()` function can be used to create vectors of objects by concatenating things together.

```
> x <- c(0.5, 0.6)        ## numeric
```

```
> x <- c(TRUE, FALSE)     ## logical
> x <- c(T, F)            ## logical
> x <- c("a", "b", "c")   ## character
> x <- 9:29               ## integer
> x <- c(1+0i, 2+4i)      ## complex
```

Note that in the above example, T and F are short-hand ways to specify TRUE and FALSE. However, in general one should try to use the explicit TRUE and FALSE values when indicating logical values. The T and F values are primarily there for when you're feeling lazy.

You can also use the vector() function to initialize vectors.

```
> x <- vector("numeric", length = 10)
> x
 [1] 0 0 0 0 0 0 0 0 0 0
```

Mixing Objects

There are occasions when different classes of R objects get mixed together. Sometimes this happens by accident but it can also happen on purpose. So what happens with the following code?

```
> y <- c(1.7, "a")     ## character
> y <- c(TRUE, 2)      ## numeric
> y <- c("a", TRUE)    ## character
```

In each case above, we are mixing objects of two different classes in a vector. But remember that the only rule about vectors says this is not allowed. When different objects are mixed in a vector, *coercion* occurs so that every element in the vector is of the same class.

In the example above, we see the effect of *implicit coercion*. What R tries to do is find a way to represent all of the objects in the vector in a reasonable fashion. Sometimes this does exactly what you want and…sometimes not. For example, combining a numeric object with a character object will create a character vector, because numbers can usually be easily represented as strings.

Explicit Coercion

Objects can be explicitly coerced from one class to another using the as.* functions, if available.

```
> x <- 0:6
> class(x)
 [1] "integer"
 >  as.numeric(x) [1] 0 1 2 3 4 5 6
> as.logical(x)
 [1] FALSE  TRUE TRUE TRUE TRUE TRUE TRUE
> as.character(x)
 [1] "0" "1" "2" "3" "4" "5" "6"
```

Sometimes, R can't figure out how to coerce an object and this can result in NAs being produced.

```
> x <- c("a", "b", "c")
> as.numeric(x)
 Warning: NAs introduced by coercion
 [1] NA NA NA
```

```
> as.logical(x)
 [1] NA NA NA
> as.complex(x)
Warning: NAs introduced by coercion
 [1] NA NA NA
```

When nonsensical coercion takes place, you will usually get a warning from R.

Matrices

Matrices are vectors with a *dimension* attribute. The dimension attribute is itself an integer vector of length 2 (number of rows, number of columns)

```
> m <- matrix(nrow = 2, ncol = 3)
> m
     [,1] [,2] [,3]
[1,]   NA   NA   NA
[2,]   NA   NA   NA
> dim(m)
[1] 2 3
> attributes(m)
$dim
[1] 2 3
```

Matrices are constructed *column-wise*, so entries can be thought of starting in the "upper left" corner and running down the columns.

```
> m <- matrix(1:6, nrow = 2, ncol = 3)
> m
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

Matrices can also be created directly from vectors by adding a dimension attribute.

```
> m <- 1:10
> m
 [1]  1  2  3  4  5  6  7  8  9 10
> dim(m) <- c(2, 5)
> m
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

Matrices can be created by *column-binding* or *row-binding* with the cbind() and rbind() functions.

```
> x <- 1:3
```

```
> y <- 10:12
> cbind(x, y)
     x y
[1,] 1 10
[2,] 2 11
[3,] 3 12
rbind(x, y)
  [,1] [,2] [,3]
x    1    2    3
y   10   11   12
```

Lists

Lists are a special type of vector that can contain elements of different classes. Lists are a very important data type in R and you should get to know them well. Lists, in combination with the various "apply" functions discussed later, make for a powerful combination.

Lists can be explicitly created using the `list()` function, which takes an arbitrary number of arguments.

```
> x <- list(1, "a", TRUE, 1 + 4i)
> x [[1]]
[1] 1

[[2]]
[1] "a"

[[3]]
[1] TRUE

[[4]]
[1] 1+4i
```

We can also create an empty list of a prespecified length with the vector() function

```
> x <- vector("list", length = 5)
> x [[1]] NULL

[[2]]
NULL

[[3]]
NULL

[[4]]
NULL
```

```
[[5]]
NULL
```

Factors

Factors are used to represent categorical data and can be unordered or ordered. One can think of a factor as an integer vector where each integer has a *label*. Factors are important in statistical modeling and are treated specially by modelling functions like lm() and glm().

Using factors with labels is *better* than using integers because factors are self-describing. Having a variable that has values "Male" and "Female" is better than a variable that has values 1 and 2.

Factor objects can be created with the factor() function.

```
> x <- factor(c("yes", "yes", "no", "yes", "no"))
> x
[1] yes yes no  yes no Levels: no yes
> table(x) x
no yes 2 3
> ## See the underlying representation of factor
> unclass(x) [1] 2 2 1 2 1
attr(,"levels")
[1] "no"  "yes"
```

Often factors will be automatically created for you when you read a dataset in using a function like read.table(). Those functions often default to creating factors when they encounter data that look like characters or strings.

The order of the levels of a factor can be set using the levels argument to factor(). This can be important in linear modelling because the first level is used as the baseline level.

```
> x <- factor(c("yes", "yes", "no", "yes", "no"))
> x  ## Levels are put in alphabetical order
[1] yes yes no  yes no Levels: no yes
> x <- factor(c("yes", "yes", "no", "yes", "no"),
+             levels = c("yes", "no"))
> x
[1]  yes yes no  yes no Levels: yes no
```

Missing Values

Missing values are denoted by NA or NaN for q undefined mathematical operations.

- is.na() is used to test objects if they are NA
- is.nan() is used to test for NaN
- NA values have a class also, so there are integer NA, character NA, etc.
- A NaN value is also NA but the converse is not true

```
> ## Create a vector with NAs in it
> x <- c(1, 2, NA, 10, 3)
```

```
> ## Return a logical vector indicating which elements are NA
> is.na(x)
 [1] FALSE FALSE TRUE FALSE FALSE
> ## Return a logical vector indicating which elements are NaN
> is.nan(x)
 [1] FALSE FALSE FALSE FALSE FALSE


> ## Now create a vector with both NA and NaN values
> x <- c(1, 2, NaN, NA, 4)
> is.na(x)
 [1] FALSE FALSE TRUE TRUE FALSE
> is.nan(x)
 [1] FALSE FALSE TRUE FALSE FALSE
```

Data Frames

Data frames are used to store tabular data in R. They are an important type of object in R and are used in a variety of statistical modeling applications. Hadley Wickham's package dplyr[35] has an optimized set of functions designed to work efficiently with data frames.

Data frames are represented as a special type of list where every element of the list has to have the same length. Each element of the list can be thought of as a column and the length of each element of the list is the number of rows.

Unlike matrices, data frames can store different classes of objects in each column. Matrices must have every element be the same class (e.g. all integers or all numeric).

In addition to column names, indicating the names of the variables or predictors, data frames have a special attribute called row.names which indicate information about each row of the data frame.

Data frames are usually created by reading in a dataset using the read.table() or read.csv(). However, data frames can also be created explicitly with the data.frame() function or they can be coerced from other types of objects like lists.

Data frames can be converted to a matrix by calling data.matrix(). While it might seem that the as.matrix() function should be used to coerce a data frame to a matrix, almost always, what you want is the result of data.matrix().

```
> x <- data.frame(foo = 1:4, bar = c(T, T, F, F))
> x
  foo   bar
1   1  TRUE
2   2  TRUE
3   3 FALSE
4   4 FALSE
> nrow(x)
 [1] 4
> ncol(x)
 [1] 2
```

Names

R objects can have names, which is very useful for writing readable code and self-describing objects. Here is an example of assigning names to an integer vector.

```
> x <- 1:3
> names(x)
  NULL
> names(x) <- c("New York", "Seattle", "Los Angeles")
> x
  New York      Seattle  Los Angeles 1       2       3
> names(x)
  [1] "New York"   "Seattle"     "Los Angeles"
```

Lists can also have names, which is often very useful.

```
> x <- list("Los Angeles" = 1, Boston = 2, London = 3)
> x
  $`Los Angeles`
  [1] 1

  $Boston
  [1] 2

  $London
  [1] 3
> names(x)
  [1] "Los Angeles" "Boston"     "London"
```

Getting Data In and Out of R

Reading and WritingData

Watch a video of this section[36]

There are a few principal functions reading data into R.

- read.table, read.csv, for reading tabular data
- readLines, for reading lines of a text file
- source, for reading in R code files (inverse of dump)
- dget, for reading in R code files (inverse of dput)
- load, for reading in saved workspaces
- unserialize, for reading single R objects in binary form

There are of course, many R packages that have been developed to read in all kinds of other datasets, and you may need to resort to one of these packages if you are working in a specific area.

There are analogous functions for writing data to files

- write.table, for writing tabular data to text files (i.e. CSV) or connections
- writeLines, for writing character data line-by-line to a file or connection
- dump, for dumping a textual representation of multiple R objects
- dput, for outputting a textual representation of an R object
- save, for saving an arbitrary number of R objects in binary format (possibly compressed) to a file.
- serialize, for converting an R object into a binary format for outputting to a connection (or file).

Reading Data Files with read.table()

The read.table() function is one of the most commonly used functions for reading data. The help file for read.table() is worth reading in its entirety if only because the function gets used a lot (run ?read.table in R). I know, I know, everyone always says to read the help file, but this one is actually worth reading.

The read.table() function has a few important arguments:

- file, the name of a file, or a connection
- header, logical indicating if the file has a header line
- sep, a string indicating how the columns are separated
- colClasses, a character vector indicating the class of each column in the dataset
- nrows, the number of rows in the dataset. By default read.table() reads an entire file.
- comment.char, a character string indicating the comment character. This defalts to "#". If there are no commented lines in your file, it's worth setting this to be the empty string "".
- skip, the number of lines to skip from the beginning
- stringsAsFactors, should character variables be coded as factors? This defaults to TRUE because back in the old days, if you had data that were stored as strings, it was because those strings represented levels of a categorical variable. Now we have lots of data that is text data and they don't always represent categorical variables. So you may want to set this to be FALSE in those cases. If you *always* want this to be FALSE, you can set a global option via options(stringsAsFactors = FALSE). I've never seen so much heat generated on discussion forums about an R function argument than the stringsAsFactors argument. Seriously.

For small to moderately sized datasets, you can usually call read.table without specifying any other arguments

```
> data <- read.table("foo.txt")
```

In this case, R will automatically

- skip lines that begin with a #
- figure out how many rows there are (and how much memory needs to be allocated)
- figure what type of variable is in each column of the table.

Telling R all these things directly makes R run faster and more efficiently. The read.csv() function is identical to read.table except that some of the defaults are set differently (like the sep argument).

Reading in Larger Datasets with read.table

Watch a video of this section[37]

With much larger datasets, there are a few things that you can do that will make your life easier and will prevent R from choking.

- Read the help page for read.table, which contains many hints
- Make a rough calculation of the memory required to store your dataset (see the next section for an example of how to do this). If the dataset is larger than the amount of RAM on your computer, you can probably stop right here.
- Set comment.char = "" if there are no commented lines in your file.
- Use the colClasses argument. Specifying this option instead of using the default can make 'read.table'

run MUCH faster, often twice as fast. In order to use this option, you have to know the class of each column in your data frame. If all of the columns are "numeric", for example, then you can just set colClasses = "numeric". A quick an dirty way to figure out the classes of each column is the following:

```
> initial <- read.table("datatable.txt", nrows = 100)
> classes <- sapply(initial, class)
> tabAll <- read.table("datatable.txt", colClasses = classes)
```

- Set nrows. This doesn't make R run faster but it helps with memory usage. A mild overestimate is okay. You can use the Unix tool wc to calculate the number of lines in a file.

In general, when using R with larger datasets, it's also useful to know a few things about your system.

- How much memory is available on your system?
- What other applications are in use? Can you close any of them?
- Are there other users logged into the same system?
- What operating system ar you using? Some operating systems can limit the amount of memory a single process can access

Calculating Memory Requirements for R Objects

Because R stores all of its objects physical memory, it is important to be cognizant of how much memory is being used up by all of the data objects residing in your workspace. One situation where it's particularly important to understand memory requirements is when you are reading in a new dataset into R. Fortunately, it's easy to make a back of the envelope calculation of how much memory will be required by a new dataset.

For example, suppose I have a data frame with 1,500,000 rows and 120 columns, all of which are numeric data. Roughly, how much memory is required to store this data frame? Well, on most modern computers double precision floating point numbers[38] are stored using 64 bits of memory, or 8 bytes. Given that information, you can do the following calculation

$1{,}500{,}000 \times 120 \times 8$ bytes/numeric = 1,440,000,000 bytes
= 1,440,000,000 / 2^{20} bytes/MB
= 1,373.29 MB
= 1.34 GB

So the dataset would require about 1.34 GB of RAM. Most computers these days have at least that much RAM. However, you need to be aware of

- what other programs might be running on your computer, using up RAM
- what other R objects might already be taking up RAM in your workspace

Reading in a large dataset for which you do not have enough RAM is one easy way to freeze up your computer (or at least your R session). This is usually an unpleasant experience that usually requires you to kill the R process, in the best case scenario, or reboot your computer, in the worst case. So make sure to do a rough calculation of memeory requirements before reading in a large dataset. You'll thank me later.

Using the readrPackage

The readr package is recently developed by Hadley Wickham to deal with reading in large flat files quickly. The package provides replacements for functions like read.table() and read.csv(). The analogous functions in readr are read_table() and read_csv(). This functions are oven *much* faster than their base R analogues and provide a few other nice features such as progress meters.

For the most part, you can read use read_table() and read_csv() pretty much anywhere you might use read.table() and read.csv(). In addition, if there are non-fatal problems that occur while reading in the data, you will get a warning and the returned data frame will have some information about which rows/observations triggered the warning. This can be very helpful for "debugging" problems with your data before you get neck deep in data analysis.

Using Textual and Binary Formats for Storing Data

Using dput() anddump()

One way to pass data around is by deparsing the R object with dput() and reading it back in (parsing it) using dget().

```
> ## Create a data frame
> y <- data.frame(a = 1, b = "a")
> ## Print 'dput' output to console
> dput(y)
structure(list(a = 1, b = structure(1L, .Label = "a", class = "factor")),
.Names\
= c("a",
"b"), row.names = c(NA, -1L), class = "data.frame")
```

Notice that the dput() output is in the form of R code and that it preserves metadata like the class of the object, the row names, and the column names.

The output of dput() can also be saved directly to a file.

```
> ## Send 'dput' output to a file
> dput(y, file = "y.R")
> ## Read in 'dput' output from a file
> new.y <- dget("y.R")
> new.y a b
1 1 a
```

Multiple objects can be deparsed at once using the dump function and read back in using source.

```
> x <- "foo"
> y <- data.frame(a = 1L, b = "a")
```

We can dump() R objects to a file by passing a character vector of their names.

```
> dump(c("x", "y"), file = "data.R")
> rm(x, y)
```

The inverse of dump() is source().

```
> source("data.R")
> str(y)
'data.frame':	1 obs. of 2 variables:
 $ a: int 1
 $ b: Factor w/ 1 level "a": 1
> x
[1] "foo"
```

Binary Formats

The complement to the textual format is the binary format, which is sometimes necessary to use for efficiency purposes, or because there's just no useful way to represent data in a textual manner. Also, with numeric data, one can often lose precision when converting to and from a textual format, so it's better to stick with a binary format.

The key functions for converting R objects into a binary format are save(), save.image(), and serialize(). Individual R objects can be saved to a file using the save() function.

```
> a <- data.frame(x = rnorm(100), y = runif(100))
> b <- c(3, 4.4, 1 / 3)
  >
> ## Save 'a' and 'b' to a file
> save(a, b, file = "mydata.rda")
  >
> ## Load 'a' and 'b' into your workspace
> load("mydata.rda")
```

If you have a lot of objects that you want to save to a file, you can save all objects in your workspace using the save.image() function.

```
> ## Save everything to a file
> save.image(file = "mydata.RData")
  >
> ## load all objects in this file
> load("mydata.RData")
```

Notice that I've used the .rda extension when using save() and the .RData extension when using save.image(). This is just my personal preference; you can use whatever file extension you want. The save() and save.image() functions do not care. However, .rda and .RData are fairly common extensions and you may want to use them because they are recognized by other software.

The serialize() function is used to convert individual R objects into a binary format that can be communicated across an arbitrary connection. This may get sent to a file, but it could get sent over a network or other connection.

When you call serialize() on an R object, the output will be a raw vector coded in hexadecimal format.

```
> x <- list(1, 2, 3)
> serialize(x, NULL)
 [1] 58 0a 00 00 00 02 00 03 02 01 00 02 03 00 00 00 00 13 00 00 00 03 00
[24] 00 00 0e 00 00 00 01 3f f0 00 00 00 00 00 00 00 00 00 0e 00 00 00 01
```

```
   ]
[47 40 00 00 00 00 00 00 00 00 00 00 0e 00 00 00 01 40 08 00 00 00 00 00
   ]
[70 00
   ]
```

If you want, this can be sent to a file, but in that case you are better off using something like save(). The benefit of the serialize() function is that it is the only way to perfectly represent an R object in an exportable format, without losing precision or any metadata. If that is what you need, then serialize() is the function for you.

Interfaces to the OutsideWorld

- file, opens a connection to a file
- gzfile, opens a connection to a file compressed with gzip
- bzfile, opens a connection to a file compressed with bzip2
- url, opens a connection to a webpage

In general, connections are powerful tools that let you navigate files or other external objects. Connections can be thought of as a translator that lets you talk to objects that are outside of R. Those outside objects could be anything from a data base, a simple text file, or a a web service API. Connections allow R functions to talk to all these different external objects without you having to write custom code for each object.

File Connections

Connections to text files can be created with the file() function.

```
> str(file)
function (description = "", open = "", blocking = TRUE, encoding =
getOption("en\ coding"),
raw = FALSE)
```

The file() function has a number of arguments that are common to many other connection functions so it's worth going into a little detail here.

- description is the name of the file
- open is a code indicating what mode the file should be opened in

The open argument allows for the following options:

- "r" open file in read only mode
- "w" open a file for writing (and initializing a new file)
- "a" open a file for appending
- "rb", "wb", "ab" reading, writing, or appending in binary mode(Windows)

In practice, we often don't need to deal with the connection interface directly as many functions for reading and writing data just deal with it in the background.
For example, if one were to explicitly use connections to read a CSV file in to R, it might look like this,

```
> ## Create a connection to 'foo.txt'
> con <- file("foo.txt")
  >
> ## Open connection to 'foo.txt' in read-only mode
> open(con, "r")
  >
> ## Read from the connection
> data <- read.csv(con)
  >
> ## Close the connection
> close(con)
```

which is the same as

```
> data <- read.csv("foo.txt")
```

In the background, read.csv() opens a connection to the file foo.txt, reads from it, and closes the connection when its done.
The above example shows the basic approach to using connections. Connections must be opened, then the are read from or written to, and then they are closed.

Reading Lines of a Text File
Text files can be read line by line using the readLines() function. This function is useful for reading text files that may be unstructured or contain non-standard data.

```
> ## Open connection to gz-compressed text file
> con <- gzfile("words.gz")
> x <- readLines(con, 10)
> x
 [1] "1080"       "10-point" "10th"   "11-point" "12-point" "16-point" [7]
 "18-point" "1st"          "2"        "20-point"
```

For more structured text data like CSV files or tab-delimited files, there are other functions like read.csv() or read.table().
The above example used the gzfile() function which is used to create a connection to files compressed using the gzip algorithm. This approach is useful because it allows you to read from a file without having to uncompress the file first, which would be a waste of space and time.
There is a complementary function writeLines() that takes a character vector and writes each element

of the vector one line at a time to a text file.

Reading From a URL Connection

The readLines() function can be useful for reading in lines of webpages. Since web pages are basically text files that are stored on a remote server, there is conceptually not much difference between a web page and a local text file. However, we need R to negotiate the communication between your computer and the web server. This is what the **url()** function can do for you, by creating a **url** connection to a web server.

This code might take time depending on your connection speed.

```
> ## Open a URL connection for reading
> con <- url("http://www.jhsph.edu", "r")
  >
> ## Read the web page
> x <- readLines(con)
  >
> ## Print out the first few lines
> head(x)
[1] "<!DOCTYPE html>"
[2]     "<html lang=\"en\">" [3] ""
[4] "<head>"
[5] "<meta charset=\"utf-8\" />"
[6] "<title>Johns Hopkins Bloomberg School of Public Health</title>"
```

While reading in a simple web page is sometimes useful, particularly if data are embedded in the web page somewhere. However, more commonly we can use URL connection to read in specific data files that are stored on web servers.

Using URL connections can be useful for producing a reproducible analysis, because the code essentially documents where the data came from and how they were obtained. This is approach is preferable to opening a web browser and downloading a dataset by hand. Of course, the code you write with connections may not be executable at a later date if things on the server side are changed or reorganized.

Subsetting R Objects

There are three operators that can be used to extract subsets of R objects.

- The **[** operator always returns an object of the same class as the original. It can be used to select multiple elements of an object
- The **[[** operator is used to extract elements of a list or a data frame. It can only be used to extract a single element and the class of the returned object will not necessarily be a list or data frame.
- The $ operator is used to extract elements of a list or data frame by literal name. Its semantics are similar to that of **[[**.

Subsetting a Vector

Vectors are basic objects in R and they can be subsetted using the **[** operator.

```
> x <- c("a", "b", "c", "c", "d", "a")
> x[1]        ## Extract the first element
  [1] "a"
> x[2]        ## Extract the second element
```

```
[1] "b"
```

The [operator can be used to extract multiple elements of a vector by passing the operator an integer sequence. Here we extract the first four elements of the vector.

```
> x[1:4]
[1] "a" "b" "c" "c"
```

The sequence does not have to be in order; you can specify any arbitrary integer vector.

```
> x[c(1, 3, 4)]
[1] "a" "c" "c"
```

We can also pass a logical sequence to the [operator to extract elements of a vector that satisfy a given condition. For example, here we want the elements of x that come lexicographically *after* the letter "a".

```
> u <- x > "a"
> u
[1] FALSE  TRUE TRUE TRUE TRUE FALSE
> x[u]
[1] "b" "c" "c" "d"
```

Another, more compact, way to do this would be to skip the creation of a logical vector and just subset the vector directly with the logical expression.

```
> x[x > "a"]
[1] "b" "c" "c" "d"
```

Subsetting a Matrix

Watch a video of this section[43]

Matrices can be subsetted in the usual way with (*i*,*j*) type indices. Here, we create simple 2×3 matrix with the matrix function.

```
> x <- matrix(1:6, 2, 3)
> x
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

We cna access the $(1, 2)$ or the $(2, 1)$ element of this matrix using the appropriate indices.

```
> x[1, 2]
[1] 3
> x[2, 1]
[1] 2
```

Indices can also be missing. This behavior is used to access entire rows or columns of a matrix.

```
> x[1, ]    ## Extract the first row
[1] 1 3 5
> x[, 2]    ## Extract the second column
[1] 3 4
```

Dropping matrix dimensions

By default, when a single element of a matrix is retrieved, it is returned as a vector of length 1 rather than a 1×1 matrix. Often, this is exactly what we want, but this behavior can be turned off by setting drop = FALSE.

```
> x <- matrix(1:6, 2, 3)
> x[1, 2]
[1] 3
> x[1, 2, drop = FALSE] [,1]
[1,]    3
```

Similarly, when we extract a single row or column of a matrix, R by default drops the dimension of length 1, so instead of getting a 1×3 matrix after extracting the first row, we get a vector of length 3. This behavior can similarly be turned off with the drop = FALSE option.

```
> x <- matrix(1:6, 2, 3)
> x[1, ] [1] 1 3 5
> x[1, , drop = FALSE] [,1] [,2] [,3]
[1,]    1    3    5
```

Be careful of R's automatic dropping of dimensions. This is a feature that is often quite useful during interactive work, but can later come back to bite you when you are writing longer programs or functions.

Subsetting Lists

Watch a video of this section[44]

Lists in R can be subsetted using all three of the operators mentioned above, and all three are used for different purposes.

```
> x <- list(foo = 1:4, bar = 0.6)
> x
$foo
[1] 1 2 3 4

$bar [1] 0.6
```

The [[operator can be used to extract *single* elements from a list. Here we extract the first element of the list.

```
> x[[1]] [1] 1 2 3 4
```

The [[operator can also use named indices so that you don't have to remember the exact ordering of every element of the list. You can also use the $ operator to extract elements by name.

```
> x[["bar"]] [1] 0.6
> x$bar [1] 0.6
```

Notice you don't need the quotes when you use the $ operator.
One thing that differentiates the [[operator from the $ is that the [[operator can be used with *computed* indices. The $ operator can only be used with literal names.

```
> x <- list(foo = 1:4, bar = 0.6, baz = "hello")
> name <- "foo"
  >
> ## computed index for "foo"
  > x[[name]] [1] 1 2 3 4
  >
> ## element "name" doesnâ€™t exist! (but no error here)
> x$name
  NULL
  >
> ## element "foo" does exist
> x$foo
  [1] 1 2 3 4
```

Subsetting Nested Elements of a List
The [[operator can take an integer sequence if you want to extract a nested element of a list.

```
> x <- list(a = list(10, 12, 14), b = c(3.14, 2.81))
  >
> ## Get the 3rd element of the 1st element
> x[[c(1, 3)]]
  [1] 14
  >
> ## Same as above
  > x[[1]][[3]] [1] 14
  >
> ## 1st element of the 2nd element
> x[[c(2, 1)]]
  [1] 3.14
```

Extracting Multiple Elements of a List
The [operator can be used to extract *multiple* elements from a list. For example, if you wanted to extract the first and third elements of a list, you would do the following

```
> x <- list(foo = 1:4, bar = 0.6, baz = "hello")
> x[c(1, 3)]
  $foo
  [1] 1 2 3 4
```

```
$baz
[1] "hello"
```

Note that x[c(1, 3)] is NOT the same as x[[c(1, 3)]].
Remember that the [operator always returns an object of the same class as the original. Since the original object was a list, the [operator returns a list. In the above code, we returned a list with two elements (the first and the third).

Partial Matching
Watch a video of this section[45]
Partial matching of names is allowed with [[and $. This is often very useful during interactive work if the object you're working with has very long element names. You can just abbreviate those names and R will figure out what element you're referring to.

```
> x <- list(aardvark = 1:5)
> x$a
[1] 1 2 3 4 5
> x[["a"]]
NULL
> x[["a", exact = FALSE]] [1] 1 2 3 4 5
```

In general, this is fine for interactive work, but you shouldn't resort to partial matching if you are writing longer scripts, functions, or programs. In those cases, you should refer to the full element name if possible. That way there's no ambiguity in your code.

Removing NA Values
Watch a video of this section[46]
A common task in data analysis is removing missing values (NAs).

```
> x <- c(1, 2, NA, 4, NA, 5)
> bad <- is.na(x)
> print(bad)
[1] FALSE FALSE TRUE FALSETRUE FALSE
> x[!bad] [1] 1 2 4 5
```

What if there are multiple R objects and you want to take the subset with no missing values in any of those objects?

```
> x <- c(1, 2, NA, 4, NA, 5)
> y <- c("a", "b", NA, "d", NA, "f")
> good <- complete.cases(x, y)
> good
[1] TRUE TRUE FALSETRUE FALSETRUE
> x[good] [1] 1 2 4 5
> y[good]
[1] "a" "b" "d" "f"
```

You can use complete.cases on data frames too.

```
> head(airquality)
```

	Ozone	Solar.R	Wind	Temp	Month	Day
1	41	190	7.4	67	5	1
2	36	118	8.0	72	5	2
3	12	149	12.6	74	5	3
4	18	313	11.5	62	5	4
5	NA	NA	14.3	56	5	5
6	28	NA	14.9	66	5	6

```
> good <- complete.cases(airquality)
> head(airquality[good, ])
```

	Ozone	Solar.R	Wind	Temp	Month	Day
1	41	190	7.4	67	5	1
2	36	118	8.0	72	5	2
3	12	149	12.6	74	5	3
4	18	313	11.5	62	5	4
7	23	299	8.6	65	5	7
8	19	99	13.8	59	5	8

Vectorized Operations

Many operations in R are *vectorized*, meaning that operations occur in parallel in certain R objects. This allows you to write code that is efficient, concise, and easier to read than in non-vectorized languages. The simplest example is when adding two vectors together.

```
> x <- 1:4
> y <- 6:9
> z <- x + y
> z
[1]  7  9 11 13
```

Natural, right? Without vectorization, you'd have to do something like

```
z <- numeric(length(x))
for(i in seq_along(x)) { z <- x[i] + y[i]
}
z
[1] 13
```

If you had to do that every time you wanted to add two vectors, your hands would get very tired from all the typing.

Another operation you can do in a vectorized manner is logical comparisons. So suppose you wanted to know which elements of a vector were greater than 2. You could do he following.

```
> x
[1] 1 2 3 4
> x > 2
[1] FALSE FALSE TRUE TRUE
```

```
> x >= 2
[1] FALSE  TRUE  TRUE  TRUE
> x < 3
[1]  TRUE  TRUE FALSE FALSE
> y == 8
[1] FALSE FALSE TRUE FALSE
```

Notice that these logical operations return a logical vector of TRUE and FALSE. Of course, subtraction, multiplication and division are also vectorized.

```
> x - y
[1] -5 -5 -5 -5
> x * y
[1]  6 14 24 36
> x / y
[1] 0.1666667 0.2857143 0.3750000 0.4444444
```

Vectorized Matrix Operations

Matrix operations are also vectorized, making for nicly compact notation. This way, we can do element-by-element operations on matrices without having to loop over every element.

```
> x <- matrix(1:4, 2, 2)
> y <- matrix(rep(10, 4), 2, 2)
>
> ## element-wise multiplication
x * y
     [,1] [,2]
[1,]   10   30
[2,]   20   40
>
> ## element-wise division
> x / y
     [,1] [,2]
[1,]  0.1  0.3
[2,]  0.2  0.4
>
## true matrix multiplication
x %*% y
     [,1] [,2]
[1,]   40   40
```

[2, 6060]

Managing Data Frames withthe dplyr package

The dplyr Package

The dplyr package was developed by Hadley Wickham of RStudio and is an optimized and distilled version of his plyr package. The dplyr package does not provide any "new" functionality to R per se, in the sense that everything dplyr does could already be done with base R, but it *greatly* simplifies existing functionality in R.

dplyr Grammar

Some of the key "verbs" provided by the dplyr package are

- select: return a subset of the columns of a data frame, using a flexible notation
- filter: extract a subset of rows from a data frame based on logical conditions
- arrange: reorder rows of a data frame
- rename: rename variables in a data frame
- mutate: add new variables/columns or transform existing variables
- summarise / summarize: generate summary statistics of different variables in the data frame, possibly within strata
- %>%: the "pipe" operator is used to connect multiple verb actions together into a pipeline

The dplyr package as a number of its own data types that it takes advantage of. For example, there is a handy print method that prevents you from printing a lot of data to the console. Most of the time, these additional data types are transparent to the user and do not need to be worried about.

Common dplyr Function Properties

All of the functions that we will discuss in this Chapter will have a few common characteristics. In particular,

1. The first argument is a data frame.
2. The subsequent arguments describe what to do with the data frame specified in the first argument, and you can refer to columns in the data frame directly without using the $ operator (just use the column names).
3. The return result of a function is a new data frame
4. Data frames must be properly formatted and annotated for this to all be useful. In particular, the data must be tidy[52]. In short, there should be one observation per row, and each column should represent a feature or characteristic of that observation.

Installing the dplyrpackage

The dplyr package can be installed from CRAN or from GitHub using the devtools package and the install_github() function. The GitHub repository will usually contain the latest updates to the package and the development version.

To install from CRAN, just run

```
install.packages("dplyr")
```

To install from GitHub you can run

```
> install_github("hadley/dplyr")
```

After installing the package it is important that you load it into your R session with the library() function.

```
> library(dplyr)

  Attaching package: 'dplyr'

The following object is masked from 'package:stats': filter
The following objects are masked from 'package:base': intersect, setdiff, setequal,
    union
```

You may get some warnings when the package is loaded because there are functions in the dplyr package that have the same name as functions in other packages. For now you can ignore the warnings.

select()

For the examples in this chapter we will be using a dataset containing air pollution and temperature data for the city of Chicago[53] in the U.S. The dataset is available from my web site.

After unzipping the archive, you can load the data into R using the readRDS() function.

```
> chicago <- readRDS("chicago.rds")
```

You can see some basic characteristics of the dataset with the dim() and str() functions.

```
> dim(chicago) [1] 6940    8
> str(chicago)
'data.frame':          6940 obs. of 8 variables:
$ city        : chr "chic" "chic" "chic" "chic" ...
$ tmpd        : num 31.5 33 33 29 32 40 34.5 29 26.5 32.5 ...
$ dptp        : num 31.5 29.9 27.4 28.6 28.9 ...
$ date        : Date, format: "1987-01-01" "1987-01-02" ...
$ pm25tmean2: num   NA NA NA NA NA NA NA NA NA NA ...
$ pm10tmean2: num   34 NA 34.2 47 NA ...
$ o3tmean2    : num 4.25 3.3 3.33 4.38 4.75 ...
$ no2tmean2 : num   20 23.2 23.8 30.4 30.3 ...
```

The select() function can be used to select columns of a data frame that you want to focus on. Often you'll have a large data frame containing "all" of the data, but any *given* analysis might only use a subset of variables or observations. The select() function allows you to get the few columns you might need.

Suppose we wanted to take the first 3 columns only. There are a few ways to do this. We could for example use numerical indices. But we can also use the names directly.

```
> names(chicago)[1:3]
 [1] "city" "tmpd" "dptp"
> subset <- select(chicago, city:dptp)
> head(subset)

  city tmpd   dptp
1 chic 31.5 31.500
2 chic 33.0 29.875
3 chic 33.0 27.375
```

```
4 chic 29.0  28.625
5 chic 32.0  28.875
6 chic 40.0  35.125
```

Note that the `:` normally cannot be used with names or strings, but inside the `select()` function you can use it to specify a range of variable names.
You can also *omit* variables using the `select()` function by using the negative sign. With `select()` you can do

```
> select(chicago, -(city:dptp))
```

which indicates that we should include every variable *except* the variables `city` through dptp. The equivalent code in base R would be

```
> i <- match("city", names(chicago))
> j <- match("dptp", names(chicago))
> head(chicago[, -(i:j)])
```

Not super intuitive, right?
The `select()` function also allows a special syntax that allows you to specify variable names based on patterns. So, for example, if you wanted to keep every variable that ends with a "2", we could do

```
> subset <- select(chicago, ends_with("2"))
> str(subset)
'data.frame':          6940 obs. of 4 variables:
$ pm25tmean2: num    NA NA NA NA NA NA NA NA NA NA ...
$ pm10tmean2: num    34 NA 34.2 47 NA ...
$ o3tmean2  : num  4.25 3.3 3.33 4.38 4.75 ...
$ no2tmean2 : num    20 23.2 23.8 30.4 30.3 ...
```

Or if we wanted to keep every variable that starts with a "d", we could do

```
> subset <- select(chicago, starts_with("d"))
> str(subset)
'data.frame':          6940 obs. of 2 variables:
$ dptp: num    31.5 29.9 27.4 28.6 28.9 ...
$ date: Date, format: "1987-01-01" "1987-01-02" ...
```

You can also use more general regular expressions if necessary. See the help page (?select) for more details.

filter()

The `filter()` function is used to extract subsets of rows from a data frame. This function is similar to the existing subset() function in R but is quite a bit faster in my experience.
Suppose we wanted to extract the rows of the chicago data frame where the levels of PM2.5 are greater than 30 (which is a reasonably high level), we could do

```
> chic.f <- filter(chicago, pm25tmean2 > 30)
> str(chic.f)
'data.frame':        194 obs. of 8 variables:
 $ city        : chr "chic" "chic" "chic" "chic" ...
 $ tmpd        : num 23 28 55 59 57 57 75 61 73 78 ...
 $ dptp        : num 21.9 25.8 51.3 53.7 52 56 65.8 59 60.3 67.1 ...
 $ date        : Date, format: "1998-01-17" "1998-01-23" ...
 $ pm25tmean2: num   38.1 34 39.4 35.4 33.3 ...
 $ pm10tmean2: num   32.5 38.7 34 28.5 35 ...
 $ o3tmean2  : num   3.18 1.75 10.79 14.3 20.66
                                          ...
 $ no2tmean2 num   25.3 29.4 25.3 31.4 26.8 ...
       :
```

You can see that there are now only 194 rows in the data frame and the distribution of the pm25tmean2 values is.

```
> summary(chic.f$pm25tmean2)

   Min. 1st Qu. Median   Mean 3rd Qu.    Max.
  30.05   32.12  35.04  36.63   39.53   61.50
```

We can place an arbitrarily complex logical sequence inside of **filter(),** so we could for example extract the rows where PM2.5 is greater than 30 *and* temperature is greater than 80 degrees Fahrenheit.

```
> chic.f <- filter(chicago, pm25tmean2 > 30 & tmpd > 80)
>      select(chic.f, date, tmpd, pm25tmean2) date tmpd pm25tmean2
1     1998-08-23   81    39.60000
2     1998-09-06   81    31.50000
3     2001-07-20   82    32.30000
4     2001-08-01   84    43.70000
5     2001-08-08   85    38.83750
6     2001-08-09   84    38.20000
7     2002-06-20   82    33.00000
8     2002-06-23   82    42.50000
9     2002-07-08   81    33.10000
10  2002-07-18     82    38.85000
11  2003-06-25     82    33.90000
12  2003-07-04     84    32.90000
13  2005-06-24     86    31.85714
14  2005-06-27     82    51.53750
15  2005-06-28     85    31.20000
16  2005-07-17     84    32.70000
17  2005-08-03     84    37.90000
```

Now there are only 17 observations where both of those conditions are met.

arrange()

The arrange() function is used to reorder rows of a data frame according to one of the variables/-columns. Reordering rows of a data frame (while preserving corresponding order of other columns) is normally a pain to do in R. The arrange() function simplifies the process quite a bit.

Here we can order the rows of the data frame by date, so that the first row is the earliest (oldest) observation and the last row is the latest (most recent) observation.

```
> chicago <- arrange(chicago, date)
```

We can now check the first few rows

```
> head(select(chicago, date, pm25tmean2), 3)
```

	date	pm25tmean2
1	1987-01-01	**NA**
2	1987-01-02	**NA**
3	1987-01-03	**NA**

and the last few rows.

```
> tail(select(chicago, date, pm25tmean2), 3)
```

	date	pm25tmean2
6938	2005-12-29	7.45000
6939	2005-12-30	15.05714
6940	2005-12-31	15.00000

Columns can be arranged in descending order too by useing the special desc() operator.

```
> chicago <- arrange(chicago, desc(date))
```

Looking at the first three and last three rows shows the dates in descending order.

```
> head(select(chicago, date, pm25tmean2), 3)
```

	date	pm25tmean2
1	2005-12-31	15.00000
2	2005-12-30	15.05714
3	2005-12-29	7.45000

```
> tail(select(chicago, date, pm25tmean2), 3)
```

	date	pm25tmean2
6938	1987-01-03	**NA**
6939	1987-01-02	**NA**
6940	1987-01-01	**NA**

rename()

Renaming a variable in a data frame in R is surprisingly hard to do! The rename() function is designed to make this process easier.

Here you can see the names of the first five variables in the chicago data frame.

```
> head(chicago[, 1:5], 3)
```

	city	tmpd	dptp	date	pm25tmean2
1	chic	35	30.1	2005-12-31	15.00000
2	chic	36	31.0	2005-12-30	15.05714
3	chic	35	29.4	2005-12-29	7.45000

The dptp column is supposed to represent the dew point temperature adn the pm25tmean2 column provides the PM2.5 data. However, these names are pretty obscure or awkward and probably be renamed to something more sensible.

```
> chicago <- rename(chicago, dewpoint = dptp, pm25 =  pm25tmean2)
> head(chicago[, 1:5], 3)
```

	city	tmpd	dewpoint	date	pm25
1	chic	35	30.1	2005-12-31	15.00000
2	chic	36	31.0	2005-12-30	15.05714
3	chic	35	29.4	2005-12-29	7.45000

mutate()

The mutate() function exists to compute transformations of variables in a data frame. Often, you want to create new variables that are derived from existing variables and mutate() provides a clean interface for doing that.

For example, with air pollution data, we often want to *detrend* the data by subtracting the mean from the data. That way we can look at whether a given day's air pollution level is higher than or less than average (as opposed to looking at its absolute level).

Here we create a pm25detrend variable that subtracts the mean from the pm25 variable.

```
> chicago <- mutate(chicago, pm25detrend = pm25 - mean(pm25, na.rm = TRUE))
> head(chicago)
```

	city	tmpd	dewpoint	date	pm25	pm10tmean2	o3tmean2	no2tmean2	pm25detrend
1	chic	35	30.1	2005-12-31	15.00000	23.5	2.531250	13.25000	-1.230958
2	chic	36	31.0	2005-12-30	15.05714	19.2	3.034420	22.80556	-1.173815
3	chic	35	29.4	2005-12-29	7.45000	23.5	6.794837	19.97222	-8.780958
4	chic	37	34.5	2005-12-28	17.75000	27.5	3.260417	19.28563	1.519042
5	chic	40	33.6	2005-12-27	23.56000	27.0	4.468750	23.50000	7.329042
6	chic	35	29.6	2005-12-26	8.40000	8.5	14.041667	16.81944	-7.830958

There is also the related transmute() function, which does the same thing as mutate() but then *drops all non-transformed variables*.

Here we detrend the PM10 and ozone (O3) variables.

```
> head(transmute(chicago,
  +                pm10detrend = pm10tmean2 - mean(pm10tmean2, na.rm = TRUE),
+                  o3detrend = o3tmean2 - mean(o3tmean2, na.rm = TRUE)))
  pm10detrend      o3detrend
1   -10.395206 -16.904263
2   -14.695206 -16.401093
3   -10.395206 -12.640676
4    -6.395206 -16.175096
5    -6.895206 -14.966763
6   -25.395206  -5.393846
```

Note that there are only two columns in the transmuted data frame.

group_by()

The group_by() function is used to generate summary statistics from the data frame within strata defined by a variable. For example, in this air pollution dataset, you might want to know what the average annual level of PM2.5 is. So the stratum is the year, and that is something we can derive from the date variable. In conjunction with the group_by() function we often use the summarize() function (or summarise() for some parts of the world).

The general operation here is a combination of splitting a data frame into separate pieces defined by a variable or group of variables (group_by()), and then applying a summary function across those subsets (summarize()).

First, we can create a year varible using as.POSIXlt().

```
> chicago <- mutate(chicago, year = as.POSIXlt(date)$year + 1900)
```

Now we can create a separate data frame that splits the original data frame by year.

```
> years <- group_by(chicago, year)
```

Finally, we compute summary statistics for each year in the data frame with the summarize() function.

```
> summarize(years, pm25 = mean(pm25, na.rm = TRUE),
  +              o3 = max(o3tmean2, na.rm = TRUE),
  +              no2 = median(no2tmean2, na.rm = TRUE)) Source: local data frame [19
  x 4]

    year      pm25       o3      no2
1   1987       NaN 62.96966 23.49369
2   1988       NaN 61.67708 24.52296
3   1989       NaN 59.72727 26.14062
4   1990       NaN 52.22917 22.59583
5   1991       NaN 63.10417 21.38194
6   1992       NaN 50.82870 24.78921
7   1993       NaN 44.30093 25.76993
8   1994       NaN 52.17844 28.47500
9   1995       NaN 66.58750 27.26042
```

10	1996	**NaN**	58.39583	26.38715
11	1997	**NaN**	56.54167	25.48143

```
12  1998  18.26467  50.66250  24.58649
13  1999  18.49646  57.48864  24.66667
14  2000  16.93806  55.76103  23.46082
15  2001  16.92632  51.81984  25.06522
16  2002  15.27335  54.88043  22.73750
17  2003  15.23183  56.16608  24.62500
18  2004  14.62864  44.48240  23.39130
19  2005  16.18556  58.84126  22.62387
```

summarize() returns a data frame with year as the first column, and then the annual averages of pm25, o3, and no2.

In a slightly more complicated example, we might want to know what are the average levels of ozone (o3) and nitrogen dioxide (no2) within quintiles of pm25. A slicker way to do this would be through a regression model, but we can actually do this quickly with group_by() and summarize().

First, we can create a categorical variable of pm25 divided into quintiles.

```
> qq <- quantile(chicago$pm25, seq(0, 1, 0.2), na.rm = TRUE)
> chicago <- mutate(chicago, pm25.quint = cut(pm25, qq))
```

Now we can group the data frame by the pm25.quint variable.

```
> quint <- group_by(chicago, pm25.quint)
```

Finally, we can compute the mean of o3 and no2 within quintiles of pm25.

```
> summarize(quint, o3 = mean(o3tmean2, na.rm = TRUE),
+                  no2 = mean(no2tmean2, na.rm = TRUE)) Source: local data frame [6 x
3]
```

```
  pm25.quint       o3     no2
1  (1.7,8.7] 21.66401 17.99129
2 (8.7,12.4] 20.38248 22.13004
3 (12.4,16.7] 20.66160 24.35708
4 (16.7,22.6] 19.88122 27.27132
5 (22.6,61.5] 20.31775 29.64427
6         NA 18.79044 25.77585
```

From the table, it seems there isn't a strong relationship between pm25 and o3, but there appears to be a positive correlation between pm25 and no2. More sophisticated statistical modeling can help to provide precise answers to these questions, but a simple application of dplyr functions can often get you most of the way there.

%>%

The pipeline operater %>% is very handy for stringing together multiple dplyr functions in a sequence of operations. Notice above that every time we wanted to apply more than one function, the sequence gets buried in a sequence of nested function calls that is difficult to read, i.e.

```
> third(second(first(x)))
```

This nesting is not a natural way to think about a sequence of operations. The %>% operator allows you to string operations in a left-to-right fashion, i.e.

```
> first(x) %>% second %>% third
```

Take the example that we just did in the last section where we computed the mean of o3 and no2 within quintiles of pm25. There we had to

1. create a new variable pm25.quint
2. split the data frame by that new variable
3. compute the mean of o3 and no2 in the sub-groups defined by pm25.quint

That can be done with the following sequence in a single R expression.

```
> mutate(chicago, pm25.quint = cut(pm25, qq)) %>%
+              group_by(pm25.quint) %>%
+              summarize(o3 = mean(o3tmean2, na.rm = TRUE),
+                        no2 = mean(no2tmean2, na.rm = TRUE)) Source: local data
frame [6 x 3]

     pm25.quint       o3       no2
1   (1.7,8.7] 21.66401 17.99129
2  (8.7,12.4] 20.38248 22.13004
3 (12.4,16.7] 20.66160 24.35708
4 (16.7,22.6] 19.88122 27.27132
5 (22.6,61.5] 20.31775 29.64427
6          NA 18.79044 25.77585
```

This way we don't have to create a set of temporary variables along the way or create a massive nested sequence of function calls.

Notice in the above code that I pass the chicago data frame to the first call to mutate(), but then afterwards I do not have to pass the first argument to group_by() or summarize(). Once you travel down the pipeline with %>%, the first argument is taken to be the output of the previous element in the pipeline.

Another example might be computing the average pollutant level by month. This could be useful to see if there are any seasonal trends in the data.

```
> mutate(chicago, month = as.POSIXlt(date)$mon + 1) %>%
  +               group_by(month) %>%
  +               summarize(pm25 = mean(pm25, na.rm = TRUE),
  +                         o3 = max(o3tmean2, na.rm = TRUE),
  +                         no2 = median(no2tmean2, na.rm = TRUE)) Source: local data
  frame [12 x 4]

    month       pm25       o3       no2 1  1  17.76996  28.22222  25.35417
  2         2  20.37513  37.37500  26.78034
  3         3  17.40818  39.05000  26.76984
  4         4  13.85879  47.94907  25.03125
  5         5  14.07420  52.75000  24.22222
  6         6  15.86461  66.58750  25.01140
  7         7  16.57087  59.54167  22.38442
  8         8  16.93380  53.96701  22.98333
  9         9  15.91279  57.48864  24.47917
  10       10  14.23557  47.09275  24.15217
  11       11  15.15794  29.45833  23.56537
  12       12  17.52221  27.70833  24.45773
```

Here we can see that o3 tends to be low in the winter months and high in the summer while no2 is higher in the winter and lower in the summer.
