R Programming Week 1

Rudy Rupp

13/10/2020

R Programming Week 1

Learning Objectives
- Install R Stoduio and Packages, including Swirl
- History of R and S
- describe differences between atomic data types
- Subset R objects using "[", "[[" and "$"
- Understand coersion, implicit and explicit
- Remove NAs from objects

Setting values

```
x <- 1  ##sets the value of x to 1
x  ## prints the value of x
```

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

```
msg <- "hello"  ##gives msg the text "hello"
msg ## prints the text set to msg
```

```
## [1] "hello"
```

You can also set calculations into the value

```
x <- x+1 ##Increases the value of x by one.
y <- 5
z <- x+y
print(z)
```

```
## [1] 7
```

The result of the expression may be auto printed.

```
x <- 5 ## nothing pinter
x  ## auto-printing occurs
```

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

```
print(x)  ## explicit printing
```

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

A vector may also be set for a variable

```
x <- 1:20  ## creates an integer sequence from 1 to 20
x
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

Calculations can also be applied on vectors

```
x <- 1:20
y <- x*3
y
```

```
##  [1]  3  6  9 12 15 18 21 24 27 30 33 36 39 42 45 48 51 54 57 60
```

Objects and Attributes

R has 5 basic or "atomic"classes of objects
- character (text)
- numeric (real numbers, ie decimal numbers)
- integer
- complex
- logical (true/false)

The most basic object is a vector.
- Vectors can contain only objects of the same class
- except *lists*

Empty vectors can be created with the vector() function. It has 2 arguments: the class of the object and the length of the vector.

Numbers

Typically treated as numeric objects (Even if "1" is entered, it is understood as "1.0000000000"). You can enter 1L to explicitly make it an integer.

Inf can be used as a number for infinity. *-Inf* also possible
NaN can be used as a missing or undefined value, e.g 0/0

Attributes

Objects can have attributes such as:
- names, dimnames
- dimensions (eg matrices and arrays)
- class
- length (the length of a vector is the number of elements in the vector)
- other, user-defined attributes

Attributes can be accessed using the *attributes()* function.

Vectors and Lists

The c() function can be used to create vectors. c can stand for concatonate.

```
x <- c(0.5, 0.6)  ## numeric
y <- c(T, F)   ## logical
z <- c("a", "b", "c")  ## character
w <- 9:29  ## integer
v <- c(1+0i, 2+4i)  ## complex

print(c(x,y,z,w,v))
```

```
##  [1] "0.5"   "0.6"   "TRUE"  "FALSE" "a"     "b"     "c"     "9"     "10"
## [10] "11"    "12"    "13"    "14"    "15"    "16"    "17"    "18"    "19"
## [19] "20"    "21"    "22"    "23"    "24"    "25"    "26"    "27"    "28"
## [28] "29"    "1+0i"  "2+4i"
```

Or use vector function:

```
x <- vector("numeric", length = 10)
x
```

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

By default for numeric vectors, the value is 0.

Mixing Objects

Objects can be forced into a different class. You won't get an error message.

```
x <- c(1.7, "a")  ## all will be treated as characters / text
x
```

```
## [1] "1.7" "a"
```

```
y <- c(TRUE, 2)  ## all will be treated as numeric. T = 1, F = 0
y
```

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

```
z <- c("a", TRUE)  ## all will be treated as characters / text
z
```

```
## [1] "a"    "TRUE"
```

You can explicitly coerce objects into another class.

```r
x <- 0:6
class(x)
```

```
## [1] "integer"
```

```r
as.numeric(x)
```

```
## [1] 0 1 2 3 4 5 6
```

```r
as.logical(x)
```

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

```r
as.character(x)
```

```
## [1] "0" "1" "2" "3" "4" "5" "6"
```

```r
as.complex(x)
```

```
## [1] 0+0i 1+0i 2+0i 3+0i 4+0i 5+0i 6+0i
```

Nonsensical coercion results in NAs

```r
x <- c("a", "b", "c")
as.numeric(x)
```

```
## Warning: NAs introduced by coercion
```

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

```r
as.logical(x)
```

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

```r
as.complex(x)
```

```
## Warning: NAs introduced by coercion
```

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

Lists

Can contain objects of different classes. Elements are encased in double brackets.

```r
x <- list(1, "a", TRUE, 1+4i)
x
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] "a"
##
## [[3]]
## [1] TRUE
##
## [[4]]
## [1] 1+4i
```

Matrices

Are vectors with a *dimension* attribute.

```r
m <- matrix(1:6, nrow = 2, ncol = 3)
m
```

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

```r
dim(m)    ## gives dimensions of matrix, rows and columns
```

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

```r
attributes(m)   ## shows attribute dim
```

```
## $dim
## [1] 2 3
```

Matrices get constructed column-wise, top to bottom.

Matrices can be made from normal vectors

```r
m <- 1:10
m
```

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

```r
dim(m) <- c(2,5)  ## adds the dimension attribute to it, while assigning a value to
 ## the dim attribute of m: the number of rows and columns.
m
```

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

Making matrices by column binding or row binding:

```
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
```

Factors

Used to represent categorical data. Ordered or unordered. You could think of a factor as an integer vector where each integer has a label. Eg: 1 - high; 2 - medium; 3 - low. They are treated specially by modeling functions like *lm()* and *glm()*. They are better than using integers because they are self-descriptive.

```
x <- factor(c("yes", "yes", "no", "yes", "no"))
x
```

```
## [1] yes yes no  yes no
## Levels: no yes
```

```
table(x)  ## counts how many of each level
```

```
## x
##  no yes
##   2   3
```

```
unclass(x)  ## removes the class of the vector, shows integer values
```

```
## [1] 2 2 1 2 1
## attr(,"levels")
## [1] "no"  "yes"
```

The order of the levels of the factor is set in alphabetical order. between "yes" and "no", No will be the baseline level. You can explicitly change this with the *level()* function.

```
x <- factor(c("yes", "yes", "no", "yes", "no"),
            levels = c("yes", "no"))
x
```

```
## [1] yes yes no  yes no
## Levels: yes no
```

Missing Values

Denoted by NA "Not Available" (or NaN "Not a Number", for undefined mathematical operations).
- *is.na()* and *is.nan()* used to test objects.
- NA values can have classes also, eg Integer NA, character NA, etc.
- NaN value is also NA, but converse is not true

```
x <- c(1, 2, NA, 10, 3)
is.na(x)
```

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

```
is.nan(x)
```

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

```
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

Used to store tabular data. - represented as a special type of list, but each element of the list has to have the same length.
- each element of the list is a column, and the length of each element is the number of rows.
- they can store different classes of objects in each column, which makes them different from matrices.
- special attribute called *row.names*
- usually created with *read.table()* or *read.csv()* or *data.frame()*
- can be converted into matrices with *data.matrix()* (will coerce objects into being the same class)

```
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
```

```r
ncol(x)
```

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

Names Attribute

Important for writing readable code and self describing objects.

```r
x <- 1:3
names(x)
```

```
## NULL
```

```r
names(x) <- c("foo", "bar", "norf")
x
```

```
##  foo  bar norf
##    1    2    3
```

```r
names(x)
```

```
## [1] "foo"  "bar"  "norf"
```

List names and matrix names:

```r
x <- list(a = 1, b = 2, c = 3)
x
```

```
## $a
## [1] 1
##
## $b
## [1] 2
##
## $c
## [1] 3
```

```r
m <- matrix(c("BB", "Bb", "Bb", "bb"), nrow = 2, ncol = 2)
dimnames(m) <- list(c("B", "b"), c("B", "b"))  ##row names, then column names
m
```

```
##   B    b
## B "BB" "Bb"
## b "Bb" "bb"
```

Reading Tabular Data

Principal functions
- *read.table()*, *read.csv()* read text files in rows and columns and return a data frame in R
- *readLines()* reads lines of a text file, gives a character vector
- *source()* read R code files
- *dget()* read R files: objects that are "deparsed"
- *load()* reading binary objects into R
- *unserialize()* also for binary objects

Analogous functions for writing data, in order:
- *write.table()*
- *writeLines()*
- *dump()*
- *dput()*
- *save()*
- *serialize()*

Reading files with read.table()

Examine *?read.table* help file. Creates a **data frame** from a file in table format. Common arguments:

- *file* name of file / path to file
- *header* indicates if the first line is a header or not
- *sep* indicates how columns are separated
- *colClasses* character vector of the same length as number of columns. Indicates class of each column (can be different for each column).
- *nrows* number of rows (not required)
- *comment.char* indicates what character indicates comments
- *skip* skips a number of lines
- *stringsAsFactors* default=true, are character variables encoded as factors?

for moderately sized datasets, you can call read.table without any arguments other than *file*. R will automatically:
- skip lines that begin with #
- find number of rows
- classify variables

However, giving R these arguments will make it runs more smoothly and predictably. *read.csv()* is an identical command except that the default separator is a comma instead of a space. CSV files are common outputs from excel files and many other programs.

Reading Larger Datasets with read.table

Things to do to facilitate reading.
- Read the Help page carefully for read.table - Make a rough calculation of the memory required to store

your dataset. Is it larger than the RAM on your computer? If so don't proceed.
- Set *comment.char = " "* if there are no commented lines in your file.
- use the colClasses argument. Specifying this option instead of using the default. R won't need to try to figure it out. If all classes are the same class you can set them all with *colClasses = "numeric"* for example. Alternatively, use the first 100 rows and let read.table find the classes, then apply them to reading the whole text:

```
initial <- read.table("datatable.txt", nrows = 100)
classes <- sapply(initial, class)  ## S apply calls the class of each column
tabAll <- read.table("datatable.txt", colClasses = classes)
```

- set *nrows* to help reduce memory usage. a mild overestimate is okay.

It's useful to know a few things:
- How much memry is available?
- what other applications are in use?
- are other users using resources of the computer?
- what operating system are you using? is it 32 or 64 bit?

Calculating Memory Requirements

Consider a data frame with 1,500,00 rows and 120 columns, all of which are numeric data. How much memory is required?

each numeric object required 8 bytes, stored in 64 bits of data.

1,500,00 x 120 x 8 bytes/numeric = 1,440,000,000 bytes
1,440,000,000 / (2^20 bytes / MB) = 1,373.29 MB
= 1.34 GB

This is the RAW data. A bit more is required to read the data: rule of thumb is you require double the memory as the raw data.

Other Text Formats

Not in form of tables. Used to store more metadata, for example class of the data in each column. Work better in version control, as text files are more easily read by git. It can be easier to fix corruption in the file.
Downside: less space-efficient.
dump() and *dput()* to write, *source()* and *dget()* to read.

```
y <- data.frame(a = 1, b = "a")
dput(y)
```

```
## structure(list(a = 1, b = "a"), class = "data.frame", row.names = c(NA,
## -1L))
```

```
dput(y, file = "y.R")
new.y <- dget("y.R")
new.y
```

```
##   a b
## 1 1 a
```

dput can only be used on one R object, but *dump* can be used in multiple.

```
x <- "foo"
y <- data.frame(a = 1, b = "a")
dump(c("x", "y"), file = "data.R")
rm(x, y)   ## removes the objects x and y
source("data.R")   ## reconstructs the objects from the file
y
```

```
##   a b
## 1 1 a
```

```
x
```

```
## [1] "foo"
```

Interfaces with other things

Establish connections with files, compressed files or other more exotic things.
- *file()*
- *gzfile()* file compressed with gzip
- *bzfile()* file compressed with bzip2
- *url()* opens a connection to a webpage

file() function has the arguments:
- *description* name of the file
- *open* has a flag to describe:
- "r" read only.
- "w" writing (and initializing a new file)
- "a" appending
- "rb", "wb", "ab", as above but in binary mode (windows)

Connections let you navigate files, too. Often connections happen behind the scenes.

```
con <- file("datatable.txt", "r")
data <- read.csv(con)
close(con)

 ## these three lines are equivalent to simply using:

data <- read.csv("datatable.txt")
```

But it can be useful to do so explicitly, such as when reading lines of a text file:

```
con <- file("datatable.txt")
x <- readLines(con, 10)
x
```

```
##  [1] "1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 "
##  [2] "9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 "
##  [3] "7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 "
##  [4] "5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 "
##  [5] "3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 "
```

```
##  [6] "1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 "
##  [7] "9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 "
##  [8] "7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 "
##  [9] "5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 "
## [10] "1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 "
```

```
close(con)
```

writeLines() take a character vector and writes each element one line at a time to a text file.
readLines() can also be used to read from a website, after establishing a connection to a non-file object.

```
con <- url("http://www.jhsph.edu", "r")
x <- readLines(con)
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>"
```

```
close(con)
```

This is another way to read data other than *read.Table* and *read.csv*.

Subsetting

extracting subsets of R objects
- "[" always returns an object of the same class as the original. e.g. subsetting a vector makes a vector. Can be used to select more than one element.
- "[[" used to extract elements of a list or dataframa. Can only extract a single element. Returned object won't necessarily be of the same class.
- "$" is used to extract elements of a list or dataframe by name. Similar to using "[["

```
x <- c("a", "b", "c", "c", "d", "a")
x[1]
```

```
## [1] "a"
```

```
x[2]
```

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

```
x[1:4]  ## subsetting using a numerical index
```

```
## [1] "a" "b" "c" "c"
```

```
x[x > "a"]  ## subsetting using a logical index
```

```
## [1] "b" "c" "c" "d"
```

```
u <- x > "a"  ## created logical vector u
u
```

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

```
x[u]  ## subsetting using a logical vector as an index
```

```
## [1] "b" "c" "c" "d"
```

Subsetting Lists

```
x <- list(foo = 1:4, bar = 0.6, baz = "hello")
x
```

```
## $foo
## [1] 1 2 3 4
##
## $bar
## [1] 0.6
##
## $baz
## [1] "hello"
```

```
x[1]  ## extract 1st element, same class as original (a list)
```

```
## $foo
## [1] 1 2 3 4
```

```
class(x[1])
```

```
## [1] "list"
```

```
x[[1]] ## extract 1st element as just a sequence
```

```
## [1] 1 2 3 4
```

```
class(x[[1]])
```

```
## [1] "integer"
```

```
x$bar ## gives element associated with name "bar"
```

```
## [1] 0.6
```

```
x[["bar"]] ## same as above
```

```
## [1] 0.6
```

```
x["bar"] ## same as x[1], returns it as a list
```

```
## $bar
## [1] 0.6
```

```
x[c(1,3)]  ## extract multiple elements, returns as same class (list)
```

```
## $foo
## [1] 1 2 3 4
##
## $baz
## [1] "hello"
```

```
name <- "foo" ## the name of an element can be saved as an object, too
x[[name]]  ## useful to look up a name that is automatically computated
```

```
## [1] 1 2 3 4
```

```
x$name  ## doesn't work as [[ ]]. Only looks for the literal name
```

```
## NULL
```

```
x$foo  ## works because "foo" is a valid name
```

```
## [1] 1 2 3 4
```

The [[can take an integer sequence:

```
x <- list(a = list(10, 12, 14), b = c(3.14, 2.81))
x[[c(1,3)]]  ## extracts the 3rd element of the 1st list
```

```
## [1] 14
```

```
             ## inception!
x[[1]][[3]]  ## double subsetting, same result as above
```

```
## [1] 14
```

```
x[[c(2,1)]]  ## one more example
```

```
## [1] 3.14
```

Subsetting Matrices

Can be subset with a row index and a column index.

```
x <- matrix(1:6, 2, 3)
x
```

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

```
x[1, 2]
```

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

```
x[2, 1]
```

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

Indices can also be missing, which will return a full row or column.

```
x[1, ]
```

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

```
x[, 2]
```

```
## [1] 3 4
```

By default, when a single element of a matrix is retrieved, it's returned as a vector of length 1, rather than a 1 x 1 matrix. this can be turned off by setting *drop = FALSE*.

```
x <- matrix(1:6, 2, 3)
x
```

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

```
x[1, 2]
```

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

```r
x[1, 2, drop = FALSE]
```

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

```r
## A similar behaviour occurs when subsetting a single column
## or row.

x[1, ]  ## returns a vector
```

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

```r
x[1, , drop = FALSE]  ## returns a matrix
```

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

Partial Matching

Partial matching names is allowed with "[[" and "$"

```r
x <- list(aardvark = 1:5, peter = 0.6, paul = 3.14)
x$a  ## by default looks for a name that matches "a".
```

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

```r
x$p  ## fails when there are multiple matches
```

```
## NULL
```

```r
x[["a"]]  ## looks for an exact match, fails
```

```
## NULL
```

```r
x[["a", exact = FALSE]]  ## default behaviour can be turned off
```

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

Removing NA values

A common task

```r
x <- c(1, 2, NA, 4, NA, 5)  ## an object that requires cleaning
bad <- is.na(x)  ## logical vector testing for NA
bad
```

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

```
x[!bad]  ## "!" used to invert the logical vector.
```

```
## [1] 1 2 4 5
```

```
         ## The logical vector determines which elements to return.
```

You can take the subset of different object that have no missing values:

```
x <- c(1, 2, NA, 4, NA, 6, 7, 8, NA, 10, 11)
y <- c("a", "b", NA, "d", NA, "f", "g", "h", "i", NA, "k")
good <- complete.cases(x, y)
good
```

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

```
x[good]  ## note some data is lost because of missing values in y
```

```
## [1]  1  2  4  6  7  8 11
```

```
y[good]  ## note some data is lost because of missing values in x
```

```
## [1] "a" "b" "d" "f" "g" "h" "k"
```

Removing NA values from dataframes. Also removes whole rows or columns where NAs are present.

```
airquality[1:6, ]  ## example data frame
```

```
##   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)
              ## logical vector tells me which rows are complete.
airquality[good, ][1:6, ]
```

```
##   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
```

```
        ## subset the airquality matrix using the logical vector
```

Vectorised Operations

Many operations in R are *vectorised*, making code more efficient, concise and easier to read.

```
x <- 1:4; y <- 6:9
x + y  ## add 1st element of x to 1st element of y, 2nd to 2nd, etc.
```

```
## [1]  7  9 11 13
```

```
x > 2  ## compares all numbers to 2
```

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

```
x >= 2
```

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

```
y == 8  ## == tests for equality
```

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

```
x * y  ## also works with other operations
```

```
## [1]  6 14 24 36
```

```
x / y
```

```
## [1] 0.1666667 0.2857143 0.3750000 0.4444444
```

Similarly with matrices:

```
x <- matrix(1:4, 2, 2); y <- matrix(rep(10, 4), 2, 2)
x
```

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

```
y
```

```
##      [,1] [,2]
## [1,]   10   10
## [2,]   10   10
```

```r
x * y  ## element-wise multiplication
```

```
##      [,1] [,2]
## [1,]   10   30
## [2,]   20   40
```

```r
x / y
```

```
##      [,1] [,2]
## [1,]  0.1  0.3
## [2,]  0.2  0.4
```

```r
x %*% y  ## true matrix multiplication
```

```
##      [,1] [,2]
## [1,]   40   40
## [2,]   60   60
```

Week 1 Quiz Questions

Part 1

```r
## Extract the subset of rows of the data frame where Ozone
## values are above 31 and Temp values are above 90. What is the
## mean of Solar.R in this subset?

quiz <- read.csv(file = "hw1_data.csv")
Ozone <- quiz[ , 1]  ## creating a list of column 1, Ozone
Temp <- quiz[ , 4]  ## creating a list of column 4, Temp
Solar.R <- quiz[ , 2]  ## creating a list of column 2, Solar.R

Truncated <- cbind(Ozone, Temp, Solar.R)  ## merge into new matrix

Cleanrows <- complete.cases(Truncated)
                ## logical vector for complete rows

ValidRows124 <- Truncated[Cleanrows, ]  ## extract complete rows

CleanOzone <- ValidRows124[ , 1]  ## List of column 1 of the new matrix
OzoneTest <- CleanOzone > 31  ## logical vector for Ozone column > 31

BigOzone <- ValidRows124[OzoneTest, ]
          ## matrix with rows where ozone is bigger than 31 extracted

CleanTemp <- BigOzone[ , 2]
          ## List of column 2 of the BigOzone matrix
TempTest <- CleanTemp > 90  ## logical vector for Temp column > 90
BigTemp <- BigOzone[TempTest, ]
        ## matrix with rows where Ozone is bigger than 31
```

```
        ## and Temp is bigger than 90
BigTemp
```

```
##       Ozone Temp Solar.R
##  [1,]    97   92     267
##  [2,]    97   92     272
##  [3,]    76   97     203
##  [4,]   118   94     225
##  [5,]    84   96     237
##  [6,]    85   94     188
##  [7,]    96   91     167
##  [8,]    78   92     197
##  [9,]    73   93     183
## [10,]    91   93     189
```

```
mean(BigTemp[ , 3])
```

```
## [1] 212.8
```

A more elegant solution:

```
## Extract the subset of rows of the data frame where Ozone
## values are above 31 and Temp values are above 90. What is the
## mean of Solar.R in this subset?

quiz <- read.csv(file = "hw1_data.csv")  ##reading file
Ozone <- quiz[ , 1]  ## list of column 1, Ozone
Temp <- quiz[ , 4]  ## list of column 4, Temp
Solar.R <- quiz[ , 2]  ## list of column 2, Solar.R

Truncated <- cbind(Ozone, Temp, Solar.R)  ## merge into new matrix

Cleanrows <- complete.cases(Truncated)
                ## logical vector for complete rows

ValidRows124 <- Truncated[Cleanrows, ]  ## extract complete rows

OandTTest <- ValidRows124[ , 1] > 31 & ValidRows124[ , 2] > 90
        ## execute both tests and save as a logical vector

ValidRows124[OandTTest, ]  ## This is the resulting matrix
```

```
##       Ozone Temp Solar.R
##  [1,]    97   92     267
##  [2,]    97   92     272
##  [3,]    76   97     203
##  [4,]   118   94     225
##  [5,]    84   96     237
##  [6,]    85   94     188
##  [7,]    96   91     167
```

```
##  [8,]    78   92     197
##  [9,]    73   93     183
## [10,]    91   93     189
```

```
mean(ValidRows124[OandTTest, ][ , 3])
```

```
## [1] 212.8
```

```
      ## taking the mean of the third column of the resulting matrix
```

Even Better:

```
## Extract the subset of rows of the data frame where Ozone
## values are above 31 and Temp values are above 90. What is the
## mean of Solar.R in this subset?

quiz <- read.csv(file = "hw1_data.csv")  ##reading file
x <- quiz[quiz$Ozone > 31 & quiz$Temp > 90,]
   ## Subset of data frame where row elements labeled "Ozone"
   ## (column label) are bigger than 31, and rows labeled
   ## "Temp" (column label) are bigger than 90.
x[!is.na(x$Ozone), ]
```

```
##     Ozone Solar.R Wind Temp Month Day
## 69     97     267  6.3   92     7   8
## 70     97     272  5.7   92     7   9
## 120    76     203  9.7   97     8  28
## 121   118     225  2.3   94     8  29
## 122    84     237  6.3   96     8  30
## 123    85     188  6.3   94     8  31
## 124    96     167  6.9   91     9   1
## 125    78     197  5.1   92     9   2
## 126    73     183  2.8   93     9   3
## 127    91     189  4.6   93     9   4
```

```
   ## subset of rows, in which th elements labeled "Ozone"
   ## are NOT NAs
mean(x[!is.na(x$Ozone),"Solar.R"]) ## mean of column "Solar.R"
```

```
## [1] 212.8
```

Part 2

```
 ## What is the mean of "Temp" when "Month" is equal to 6?
quiz <- read.csv(file = "hw1_data.csv")  ##reading file
Temp <- quiz[ , 4]  ## list of column 4, Temp
Month <- quiz[ , 5]  ## list of column 5, Month
```

```r
Truncated <- cbind(Temp, Month)  ## merge into new matrix
Cleanrows <- complete.cases(Truncated)
                ## logical vector for complete rows
ValidRows124 <- Truncated[Cleanrows, ]  ## extract complete rows

June <- ValidRows124[ , 2] == 6  ## Logical vector for month = 6
ValidRows124[June, ]  ## the matrix with extracted rows only
```

```
##       Temp Month
##  [1,]   78     6
##  [2,]   74     6
##  [3,]   67     6
##  [4,]   84     6
##  [5,]   85     6
##  [6,]   79     6
##  [7,]   82     6
##  [8,]   87     6
##  [9,]   90     6
## [10,]   87     6
## [11,]   93     6
## [12,]   92     6
## [13,]   82     6
## [14,]   80     6
## [15,]   79     6
## [16,]   77     6
## [17,]   72     6
## [18,]   65     6
## [19,]   73     6
## [20,]   76     6
## [21,]   77     6
## [22,]   76     6
## [23,]   76     6
## [24,]   76     6
## [25,]   75     6
## [26,]   78     6
## [27,]   73     6
## [28,]   80     6
## [29,]   77     6
## [30,]   83     6
```

```r
mean(ValidRows124[June, ][ , 1])
```

```
## [1] 79.1
```

```r
    ## mean of the first column of the matrix with extracted rows only
```

Better

```r
 ## What is the mean of "Temp" when "Month" is equal to 6?
quiz <- read.csv(file = "hw1_data.csv")  ##reading file
```

```
quiz[quiz$Month == 6, "Temp"]
```

```
##  [1] 78 74 67 84 85 79 82 87 90 87 93 92 82 80 79 77 72 65 73 76 77 76 76 76 75
## [26] 78 73 80 77 83
```

```
  ## Subset elements of column "Temp", on rows where elements
  ## labeled "Month" are equal to 6.
mean(quiz[quiz$Month == 6, "Temp"])
```

```
## [1] 79.1
```

Part 3

```
## What was the maximum ozone value in the month of May
## (i.e. Month is equal to 5)?
quiz <- read.csv(file = "hw1_data.csv")  ##reading file
Ozone <- quiz[ , 1]  ## list of column 1, Ozone
Month <- quiz[ , 5]  ## list of column 5, Month

Truncated <- cbind(Ozone, Month)  ## merge into new matrix
Cleanrows <- complete.cases(Truncated)
                ## logical vector for complete rows
ValidRows124 <- Truncated[Cleanrows, ]  ## extract complete rows

May <- ValidRows124[ , 2] == 5  ## Logical vector for month = 5
ValidRows124[May, ]  ## the matrix with extracted rows only
```

```
##       Ozone Month
##  [1,]    41     5
##  [2,]    36     5
##  [3,]    12     5
##  [4,]    18     5
##  [5,]    28     5
##  [6,]    23     5
##  [7,]    19     5
##  [8,]     8     5
##  [9,]     7     5
## [10,]    16     5
## [11,]    11     5
## [12,]    14     5
## [13,]    18     5
## [14,]    14     5
## [15,]    34     5
## [16,]     6     5
## [17,]    30     5
## [18,]    11     5
## [19,]     1     5
## [20,]    11     5
## [21,]     4     5
## [22,]    32     5
```

```
## [23,]    23     5
## [24,]    45     5
## [25,]   115     5
## [26,]    37     5
```

```
max(ValidRows124[May, ][ , 1])
```

```
## [1] 115
```

```
    ## mean of the first column of the matrix with extracted rows only
```

Better

```
## What was the maximum ozone value in the month of May
## (i.e. Month is equal to 5)?
quiz <- read.csv(file = "hw1_data.csv")  ##reading file

quiz[quiz$Month == 5, "Ozone"]
```

```
##  [1]  41  36  12  18  NA  28  23  19   8  NA   7  16  11  14  18  14  34   6  30
## [20]  11   1  11   4  32  NA  NA  NA  23  45 115  37
```

```
  ## extracts the column "Ozone" from the rows where elements
  ## labeled "Month" are equal to 5
  ## NAs are still present

quiz[quiz$Month == 5 & !is.na(quiz$Ozone), "Ozone"]
```

```
##  [1]  41  36  12  18  28  23  19   8   7  16  11  14  18  14  34   6  30  11   1
## [20]  11   4  32  23  45 115  37
```

```
  ## add onto that a logical vector to remove NAs

max(quiz[quiz$Month == 5 & !is.na(quiz$Ozone), "Ozone"])
```

```
## [1] 115
```

Swirl Exercises Week 1

Swirl lesson 1: Basic Building Blocks

New commands:

- *ls()* displays a list of all variables saved in the workspace
- *getwd()* displays the path t the working directory

- pressing the up arrow on the console shows the previous command
- *c()* can be used to merge vectors or lists, eg:

```
x <- c(2, 23, 1)
c(x, 21, x)
```

```
## [1]  2 23  1 21  2 23  1
```

Swirl lesson 2: Workspace and files

- *list.files()* displays a character vector with the names of files in the directory.
  – *path =* argument to choose directory. Default is *getwd()*
  – *?list.files* help page has useful information on arguments. – *file.info()*, *file.access()*, *files()*, *file.choose()* and *choose.files()* could be useful.
- *args()* with the function name in the brackets, shows all arguments a function can take, and their default values.
- *dir.create("testdir")* to create a new folder.
- *setwd("testdir")* sets a new working directory.
- *file.create("mytest.R")* creates new file.
- *file.exists("mytest.R")* logical vector, tests if file exists.
- *file.info("mytest.R")* returns a data frame with details of size, directory, mode, and time stamps.
  – *file.info("mytest.R")$mode* retrieves specific items from the data frame
- *file.rename("mytest.R","mytest2.R")*
- *file.copy("mytest2.R", "mytest3.R")*
- *file.path("folder1", mytest3.R")* Character vector that holds a path to a file from components, in a platform-independent way. Example returns *"folder1/mytest3.R"*
  – for example *dir.create(file.path("testdir2", "testdir3"), recursive = TRUE)* creates folder"testdir3" inside folder "testdir2".

Swirl lesson 3: Sequences of Numbers

- *1:20* gives a numerical vector of length 20. Numbers start at 1 and finish at 20, increments of 1.
  – *pi:10* also works. value of 10 is not reached.
- *seq(1, 10)* does the same, but has more arguments.
  – *seq(1, 10, by=0.5)* gives increments of 0.5
  – *seq(5, 10, length=30)* calculates even intervals to match the length.
  – *seq(along.with = my_seq)* along.with take the length from the length of this argument.
- *length()* gives the length of a vector.

- *rep(0, times = 40)* replicates a value to create a vector.
  – can also take a vector value. *rep(c(0, 1, 2), times = 10)*
  – alternatively, repeat each value with *rep(c(0, 1, 2), each = 10)*

Swirl lesson 4: Vectors

- logical operators >, <, >=, <=, == and !=
- "A | B" means A OR B. "A & B" means A AND B
- !A is the negation of A and is TRUE when A is FALSE and vice versa.
- *paste(my_char, collapse = " ")* merges the elements of a vector, separating them with a space.
- can also be used to join multiple character vectors:

```
paste(1:3, c("X", "Y", "Z"), sep = "")
```

```
## [1] "1X" "2Y" "3Z"
```

```
      ## the numeric vector 1:4 gets 'coerced' into
      ## a character vector by the paste() function.
paste(LETTERS, 1:4, sep = "-")
```

```
##  [1] "A-1" "B-2" "C-3" "D-4" "E-1" "F-2" "G-3" "H-4" "I-1" "J-2" "K-3" "L-4"
## [13] "M-1" "N-2" "O-3" "P-4" "Q-1" "R-2" "S-3" "T-4" "U-1" "V-2" "W-3" "X-4"
## [25] "Y-1" "Z-2"
```

Swirl lesson 5: Missing Values

- *rnorm(1000)* to create a vector with 1000 draws from a standard normal distribution.
- *sample(y, 100)* takes 100 values at random from vector y.

Swirl lesson 6: Subsetting Vectors

- *x[!is.na(x)]* shorthand to remove NAs
- *x[c(3, 20, 21)]* picks the 3rd, 20th and 21st elements of x
- *x[-c(3, 20, 21)]* picks all elements of x, except the 3rd, 20th and 21st.
- **Named elements in vectors**

```
c(foo = 11, bar = 2, norf = NA)  ## named from start
```

```
##  foo  bar norf
##   11    2   NA
```

```r
vect <- c(11, 2, NA)
names(vect) <- c("foo", "bar", "norf")  ## named after existing
vect
```

```
##  foo  bar norf
##   11    2   NA
```

- *identical (x, y)* gives a logical vector of length 1

Swirl lesson 7: Matrices and Data Frames

- *dim(x)* gives the dimensions of an object. For a vector, that's NULL. Rows, then Columns.
- *length(x)* works for vectors.
- giving dimensions to a vector will make it into a matrix
- *colnames()* can call or set the names of columsn in a data frame
- *colnames(my_data) <- c("patient", "age", "weight", "bp", "rating", "test")*