R Programming

Coursera Course by John Hopkins University

INSTRUCTORS: Dr. Jeff Leek, Dr. Roger D. Peng, Dr. Brian Caffo

Contents

Overview of R, R data types and objects, reading and writing data

Installing R & RStudio

- This was covered in the previous course.

R-Markdown reference site

- I found a site that expands on some features of R-Markdown and have been referencing it pretty regularly

Swirl

- swirl teaches you R programming and data science interactively, at your own pace, and right in the R console.

- Start swirl
  - install the package "swirl" if you haven't yet
  - Every time you want to run swirl execute:
    - library("swirl")
    - swirl()
  - You'll then be prompted to install a course
  - Help page for swirl

History of S and R programming

- What is S?
  - R is a dialect of S
  - S was developed by John Chambers and others at Bell Labs
  - Initiated in 1976 as an internal statistical analysis environment, implemented as FORTRAN libraries
    - Early versions did not contain functions for statistical modeling
  - Version 3 was released in 1988, which was rewritten in C and began to resemble the system that we have today.
  - Version 4 was released in 1998 and is the version we use today.
    - This version is documented in *Programming with Data* by John Chambers (the green book)

  - Insightful sells its implementation of the S language under the name *S-PLUS*, which includes a number of fancy features, mostly GUIs.
  - S won the Association for Computing Machinery's Software System Award in '98

  - (More about S)[https://web.archive.org/web/20181014111802/ect.bell-labs.com/sl/S/]
- What is R?
  - R was developed by Ross Ihaka and Robert Gentleman, they documented their experience in a (1996 JCGS paper)[https://amstat.tandfonline.com/doi/abs/10.1080/10618600.1996.10474713].

  - In 1995, R become free software after Martin Machler convinced Ross & Robert to use the GNU (General Public License)

  - Versions
    * R version 1.0.0 was released in 2000

    * R version 3.0.2 is released in Dec. 2013

  - Syntax is similar to S, making it easy for S-PLUS users to switch over

  - Runs on almost any standard computing platform/OS (even on the PS3)

  - Frequent releases; active development and communities

  - Functionality is divided into modular packages as to keep it "lean"

  - It's free!

  - What is free about Free Software?
    * Freedom 0: freedom to run the program, for any purpose

    * Freedom 1: freedom to study how the program works, and adapt it to one's needs. Which implies access to the source code
    * Freedom 2: freedom to redistribute copies
    * Freedom 3: freedom to improve the program, and release your improvements to the public, or to sell them.
    * These are outlined by the (Free Software Foundation)[https://www.fsf.org/]
- Drawbacks of R
  - Essentially based on 40 year old technology,the original S language
  - Little build support for dynamic or 3D graphics. Although there are packages for such
  - Functionality is based on consumer demand and use contributions, if a feature is not present you'll have to build it.

  - Objects that are manipulated in R have to be stored in the physical memory of the computer, as such if an object is bigger than the memory you'll be unable to load it into memory

  - Not ideal for all possible situations, such as calling to order pizza (but this is a drawback

of all software packages)

*Design of the R System
+ "base" R system that can be downloaded from (CRAN)[http://cran.r-project.org] (krey-an) which...
- contains the packages: **utils, stats, datasets, graphics, grDevices, grid, methods, tools, parallel, compiler, splines, tcltk, stats4**.
- and "Recommends" the packages: **boot, class, cluster, codetools, foreign, KernSmooth, lattice, mgcv, nlme, rpart, survival, MASS, spatial, nnet, Matrix**.
+ Packages are available all around the web, but packages on CRAN have to meet a certain level of quality.

- Some Useful Books on S/R
  - Chambers (2008). *Software for Data Analysis*, Springer.
  - Chambers (1998). *Programming with Data*, Springer.
  - Venables & Ripley (2002). *Modern Applied Statistics with S*, Springer.
  - Venables & Ripley (2000). *S Programming*, Springer.
  - Pinheiro & Bates (2000). *Mixed-Effects Models in S and S-Plus*, Springer.
  - Murrell (2005). *R Graphics*, Chapman & Hall/CRC Press.
  - (Additional Books)[http://www.r-project.org/doc/bib/R-books.html]

Review of getting help

- Covered in previous course

Input and Evaluation: Vocabulary/Syntax

- **Expressions -** The code that is typed into the R prompt.
- **Assignment Operator -** assigns a value to a symbol, Ex:
  `x <- 1`
- *Output a variable*:

```
x <- 36
print(x) ##explicit printing
```

```
## [1] 36
```

```
## or one can just type the variable
x ##auto-printing
```

```
## [1] 36
```

- *Comment:* Use a Hash(#) symbol to make a comment to the right of #

- *[1]* is indicating the following variable is the first element of the vector

```
x <- 1:30 ##Loads x with the numbers 1 to 30
print(x)
```

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

```
## here, [26] is telling you the next number is the 26th element of the vector
```

- `Inf` - represents infinity and can be used in ordinary calculations (Ex: 1 / `Inf` is 0)

- `Nan` - represents an undefined value ("not a number") (Ex: 0/0 is `NaN`).
  - Can also be thought of as a missing value

- **Attributes -** Some objects in R come with attributes. These attributes can be set or modified with the expression `attributes()`. They are:
  - names, dimnames (dimension names)
  - dimensions (e.g. matrices, arrays) - number of rows & cols, or more depending on dimensions of array
  - class - the data type of the object
  - length - number of elements
  - other user-defined attributes/metadata can be added

- **Coercion -** occurs so that every element of a vector is of the same class (Covered further in Vector section)

Different atomic data types

- R has five basic, or "atomic", classes of objects:
  - character
    - In `R` there is no `string` data type. It is also considered part of the `character` data type
  - numeric (real numbers)
    - R thinks as numbers as these by default
  - integer
    - Must be explicitly declared with the L suffix; `x <- 1` assigns a numeric object, but `x <- 1L` explicitly assigns an integer
  - complex
  - logical (True/False)
- A vector can only contain objects of the same class
  - an empty vector can be created with `vector()`

- However, a **list** is represented as a vector but can contain objects of different classes (as such we usually use these)

Vectors, Lists, and Matrices

- The `c()` function (can be thought to stand for "concatenate")
  - Can be used to create vectors of objects

```
x <- c(0.5, 0.6) ## numeric
x <- c(TRUE, FALSE) ## logical
x <- c(T, F) ## logical
x <- c("a", "b", "c") ## character
x <- c(1+0i, 2+4i) ## complex
```

- The `vector()` function
  - Can also be used to create, you guessed it, vectors

```
x <- vector() ## Creates an empty vector
x ## Prints as code that evaluates as FALSE
```

```
## logical(0)
```

```
x <- vector(mode = "numeric", length = 10)
## Creates a vector with length "10" of numeric data type,default value is 0
x
```

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

```
x <- vector("numeric", 5)
##The parameter names are not required, but can easily clarify code
x
```

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

- When different objects are mixed in a vector, **coercion** occurs so all objects are of the same class.
  - R will implicitly create the "Least Common Denominator" of the mixed classes

```
y <- c(1.7, "a") ## character
y
```

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

```
y <- c(TRUE, 2) ## numeric
y
```

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

```
y <- c("a", TRUE) ## character
y
```

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

```
y[2] ## "TRUE" is a string stored as a "character" data type
```

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

```
y[3] ## The third element does not exist
```

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

- Objects can be **explicitly coerced** from one class to another using the `as.*` functions, if available.
  - Nonsensical coercion results in `NA`s

```
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"
```

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

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

```
x
```

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

```
y <- as.character(x)
y
```

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

```
x <- c("a", "b", "c")
as.numeric(x) ##Nonsensical coercion will also show a warning
```

```
## 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
```

- Lists (Important data type in R that you should get to know well)
  - Lists are a type of vector that can contain elements of different classes.
  - Doesn't print like a vector because every element is different
    - prints index of element with double brackets bordering it: `[[1]]`

```
x <- list(1, "a", TRUE, 1 + 4i, 16 +18i)
x
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] "a"
##
## [[3]]
## [1] TRUE
##
## [[4]]
## [1] 1+4i
##
## [[5]]
## [1] 16+18i
```

- **Matrices -** a type of vector with a *dimension* attribute.
  - The *dimension* attribute is itself an integer vector of length 2 (numRows, numCols)
  - Constructed *column-wise*, so entries can be thought of starting in the "upper left" corner, then running down the columns
  - Matrices can also be created by adding a *dimension* attribute to an existing vector

```
m <- matrix(nrow = 2, ncol = 3)
m
```

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

```
dim(m)##reports num of rows then cols
```

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

```
attributes(m) ## dim is an attribute of the vector
```

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

```
m <- matrix(1:6, 2, 3) ## Demonstrating column-wise filling of matrix
m
```

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

```r
m <- 1:10 ## m is now just a vector
m
```

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

```r
dim(m) <- c(2,5) ## adding the dimension attribute
m
```

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

- Creating a matrix with **cbind** and **rbind**
  - cbind fills the columns with the elements of the vectors that are passed as the respective parameters
  - likewise, rbind fills the rows with the elements of the respective parameters

```r
x <- 1:3
y <- 10:12
cbind(x,y)
```

```
##      x  y
## [1,] 1 10
## [2,] 2 11
## [3,] 3 12
```

```r
rbind(x,y)
```

```
##   [,1] [,2] [,3]
## x    1    2    3
## y   10   11   12
```

Other data types

- Factors
  - Used to represent categorical data
  - can be unordered or ordered
  - Kinda like enumerated data, where it's an integer at heart, and each integer has a *label*
  - Using factors with labels is *better* than using integers because factors are self-describing
    - consider "Male" and "Female" as opposed to just the values 1 and 2
  - Prints differently than a character value, does not include quotations and displays *Levels*

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

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

```r
table(x)
```

```
## x
```

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

```
## displays a frequency table of the factors
unclass(x)
```

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

```
## strips out the class and displays the underlying integer vector
```

- The order of the levels can be set with the `levels` argument to `factor()`
  - This can be important in linear modelling because the first level is sued as the baseline level.
  - default levels are based alphabetically

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

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

- Missing Values (`NA` or `NaN`)
  - `NaN` is for undefined mathematical operations
  - `is.na()` and `is.nan()` are logical tests for the respective missing values
  - `NA` values have a class also, so there are integer `NA`, character `NA`, etc.
  - a `NaN` is also a `NA`, however the 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
  - Special type of list where every element has to have the same length
  - Each element is like a column and the length of each element is the number of rows
  - like lists, Data Frames can store different classes in each column

  - Attribute: `row.names`
    - Useful for annotating data
    - However, often the row names are not interesting and we use "1, 2, 3..."
  - Usually created by calling `read.table()` or `read.csv()`
  - Can be converted to a matrix with `data.matrix()`
    - Forces each object to be coerced

```
x <- data.frame(foo = 1:4, bar = c(T, T, F, F))## cols are named here
x
```

```
##   foo   bar
## 1   1  TRUE
## 2   2  TRUE
## 3   3 FALSE
## 4   4 FALSE
```

```
nrow(x)
```

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

```
ncol(x)
```

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

```
row.names(x)
```

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

- Names Attribute, useful for writing readable code and self-describing objects
  - Any R object can have names

```
x <- 1:3
names(x)## by default there are no names
```

```
## NULL
```

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

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

```
names(x)
```

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

```
##Lists can also have names
x <- list(a=1, b=2, c=3)
## here, names are assigned as list is established
x
```

```
## $a
## [1] 1
##
## $b
```

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

```
## Matrices can also have names, called dimnames
m <- matrix(1:4, nrow = 2, ncol = 2)
m
```

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

```
dimnames(m) <- list(c("a", "b"), c("c", "d"))
#First vector is rownames, second is colnames
m
```

```
##   c d
## a 1 3
## b 2 4
```

Reading Data

Tabular Data

- Functions for **reading** data into R
  - `read.table`,`read.csv` - for reading tabular data
    - most common
    - reads in data that's organized into rows and cols
    - returns a data frame
  - `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
- Functions for **writing** data from R to files
  - `write.table`
  - `writeLines`
  - `dump`

  - `dput`
  - `save`
  - `serialize`

- Arguments of `read.table` function
  - `file` - the name of a file or connection
  - `header` - logical that indicates if the file has a header line
  - `sep` - a string that indicates how the columns are separated (tokens)
  - `colClasses` - a character vector that indicates the class (Data type) of each column
  - `nrows`
  - `comment.char` - character string that indicates the comment character (default is '#')
  - `skip` - number of lines to skip from the beginning
  - `stringsAsFactors` - (default = TRUE) should character variables be coded as factors?

- Implicit actions `R` takes

```
data <- read.table("foo.txt")
## Header must not have a label for the row labels for R to implicitly determine them
data
```

```
##             Price Num_Sold In_Stock Complex_Num
## Chips&Salsa  2.55     1729     TRUE       1+ 2i
## Drink        1.99     3435     TRUE       5+18i
## Taco         3.49       36    FALSE       3+ 0i
```

- Skips lines that begin with a #
- figures 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 will make it run faster and more efficiently
- `read.csv` is identical to `read.table` except that the default separator is a comma
  - `.csv` files are common output from excel or other spreadsheet programs.

Large Data-sets

- Doing the following things will make your life easier and 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 data-set.
    * Say for example, you have a data frame with 1,500,000 rows and 120 columns (not

that big), all of which are numeric data. To roughly calculate how much memory is required..

    * 1,500,000 * 120 * 8 bytes/numeric = 1440000000 bytes
    * 1440000000 bytes / 2^{20} bytes/MB = 1,373.29 MB
    * 1,373.29 MB = 1.37 GB
    * Rule of thumb is that you'll need twice the amount of RAM to be able to read in the data-set

  - If the data-set is larger than the amount of RAM on your computer you can probably stop right here.
    * Type `free -k` in terminal to return amount of RAM in kilobytes (`-b` for bytes, `-m` for megabytes and `-g` for gigabytes)
  - 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.
    * To use this option you have to know the class of each column in your data frame.
    * If all of the columns are of the same data type, for example "numeric", then you can just set `colClasses = "numeric"`
    * A quick and dirty way to figure out the classes of each column is to take a small sample and determine it from that.

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

```
##     Index   Boolean
## "integer" "integer"
```

```
##Coded file wrong, I'm not gonna fix it now, Boolean should have said "TRUE" or "FALSE"
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 type `wc <filename>` in terminal to return the number of: lines, strings, characters; "lines" are the `nrows`.

Useful things to know about your system when using R with larger data-sets

- How much memory is available
  - Type `free -k` into terminal
- What other applications are in use

  - Type `ps aux` in terminal

- Are there other users logged into the same system
  - Type `w` in terminal (Note: `last` will report a history)

- What OS are you using
  - Type `lsb_release -a` into terminal

- Is the OS 32 or 64 bit
  - Type `lscpu`, listed under first two returns
  - On a 64 bit system you'll generally be able to access more memory

Textual Formats

- Contains the metadata, such as classes of columns, making transferring data more efficient as the metadata doesn't need to be determined again.

- Known as `dumping` and `dputing`.
- Edit-able, which in the case of corruption allows for a potential recovery.
- Textual formats can work much better with version control programs.
- Adhere to the "Unix philosophy", which is to store data as text
- *Downside:* The format is not very space-efficient and as such usually requires compression

- `dput` will deparse an R object, and `dget` can read the data back in from a file

```
y <- data.frame(a=1, b="a")
dput(y) ## If file is not specified the output is displayed in the console
```

```
## structure(list(a = 1, b = structure(1L, .Label = "a", class = "factor")), class = "data.fra
## -1L))
```

```
dput(y, file = "y.R")
new.y <- dget("y.R")##dget retrieves the object from a file
new.y
```

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

- Multiple objects can be deparsed using the `dump` function, then read back in with `source`
  - The parameter for `dump` is a character vector that contains characters for the names of the variables one wishes to dump

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

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

```
x
```

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

Connections (Interfaces to the outside world)

- Connections can be made to files or to other, more "exotic" things.
  - `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 HTML format).
- Arguments
  - `description` is the name of the file
  - `open` indicates how the file is opened
    - **"r" -** read only
    - **"w" -** writing (and initializing a new file)
    - **"a" -** appending
    - **"rb", "wb", "ab" -** reading, writing, or appending in binary mode (Windows)
    - There are other options but they aren't uber important
- Connections are powerful tools that allow you to navigate files or other external objects in a more "sophisticated" way.
  - However, one does not need to deal with the connection interface in many case

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

- ^This is the same as..

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

- As such, the connection was not necessary for this case
- Reading lines of a text file with `con` from a *gzip* file

```
con <- gzfile("words.gz")
x <- readLines(con, 10) ##reads in first 10 lines
x
```

```
##  [1] "1080"     "10-point" "10th"     "11-point" "12-point" "16-point"
##  [7] "18-point" "1st"      "2"        "20-point"
```

- `writeLines` takes a character vector and writes each element one line at a time to a text file

- `readLines` can be used for reading in lines of webpages.

```
## This might take time
con <- url("http://www.jhsph.edu", "r") ##John Hopkin's School of Public Health
x <- readLines(con)
head(x) ##Displays the header
```

```
## [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>"
```

Subsetting R objects using the "[", "[[", and "$" operators and logical vectors

Basics

- Operators to extract subsets of R objects
  - `[` always returns an object of the same class as the original
    - subsetting a vector will return a vector, a list will return a list, etc.
    - Can be used to select more than one element (there is one exception, when subsetting a single element from a matrix)
  - `[[` is used to extract elements of a *list* or *data frame*
    - Can only be used to extract a single element
    - The class of the returned object will not necessarily be a list or data frame
  - `$` is used to extract elements of a *list* or *data frame* by name
    - Similar to `[[` as it may not be of the same class
- Numerical Index for subsetting:

```
x <- c("a", "b", "c", "c", "d", "a")
x[1] ## Returns first element
```

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

```
x[2] ## Returns second element
```

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

```
x[1:4] ## Returns first to fourth elements
```

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

```
x[c(2, 5)] ##Returns 2nd and 5th element
```

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

```
x[c(-2, -5)]##Returns everything EXCEPT the 2nd and 5th element
```

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

```
x[-c(2,5)]##Equivelent since the negative will multiply with every element of c(...)
```

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

```
x[2*c(1,3)]##Just like how this will actually be the 2nd and 6th element
```

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

- Logical Index for subsetting:

```
x <- c("a", "b", "c", "c", "d", "a")
x[x > "a"]## returns all elements that are greater than "a"
```

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

```
u <- x > "a"
## u is a logical vector that indicates which elements of x are greater than "a"
u
```

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

```
x[u]
```

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

```
## subsets all elements of x such that u reports that index as TRUE;
##elements that are > "a"
```

Lists

- Lists can be subsetted with the `[[` or `$` operators

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

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

```
##Extracts the first element as a list, since the orginal set was a list class
x[[1]]##Extracts the first element as a sequence, not a list
```

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

```
x$bar ##returns the element that is associated with the name "bar"
```

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

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

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

```
x["bar"]##returns a list with the element "bar" in it
```

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

- subsetting with the name is helpful when the index isn't known
- To extract multiple elements of a list, one must use the single bracket operator `[`

```
x <- list(foo = 1:4, bar = 0.6, baz = "hello")
x[c(1, 3)]##extracts the first and third element of the list
```

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

- The `[[` operator can be used with *computed* indices, whereas `$` can only be used with literal names.

```
x <- list(foo = 1:4, bar = 0.6, baz = "hello")
name <- "foo"
x[[name]] ## computed index for 'foo'
```

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

```
x$name ## element 'name' doesn't exist!
```

```
## NULL
```

```
x$foo ## element 'foo' does exist
```

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

- The `[[` can also take an integer sequence instead of a single number

```
x <- list(a = list(10, 12, 14), b = c(3.14, 2.81))
x[[c(1,3)]]
```

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

```
##extracts first element, then the third element of said first element
x[[1]][[3]] ##equivelent
```

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

```
x[[c(2,1)]]##extracts first element of the second element of x
```

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

Matrices

- Subsetted as one would expect with (i,j) type indices.

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

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

```r
x[1,2] ##First row, second column
```

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

```r
x[2,1] ##Second row, first column
```

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

- Indices can also be missing

```r
x[1,] ##Returns first row
```

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

```r
x[,2] ##Returns second column
```

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

- By default, when a single element of a matrix is retrieved, it is returned as a vector of length 1 rather than a 1x1 matrix.
  - This is the exception of the `[` operator always returning the same class
  - This behavior can be turned off with the setting `drop = FALSE`.

```r
x <- matrix(1:6, 2, 3)
x[1,2] ##returns vector
```

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

```r
x[1,2, drop = FALSE] ##returns a 1x1 matrix
```

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

- This transition of classes also holds when subsetting a single column or row

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

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

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

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

```r
## the second parameter still has to be blank so the row is returned
```

Partial Matching

- Allows one to not type out the full name of an element
  - Works with the [[and $ operators

```
x <- list(aardvark = 1:5, baking = 1:10)
x$a
```

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

```
##$ looks for a name that matches the "a", since aardvark
##starts with an "a" that is returned
x[["a"]]
```

```
## NULL
```

```
x[["a",exact = FALSE]]
```

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

```
## exact parameter has to be set to
##false for the [[ to accept a partial match

y <- list(aardvark = 1:5, apples = 1:3)
y$a
```

```
## NULL
```

```
y[["a", exact = FALSE]]
```

```
## NULL
```

```
## Since there are two names that start with "a" the intended
##element cannot be determined and NULL is returned
```

Removing missing (NA) values from a vector

- A common operation that needs to be done IRL data

```
x <- c(1, 2, NA, 4, NA, 5)
bad <- is.na(x) ## Creates a logical vector that is TRUE if the
##element is missing, and FLASE if the element is not missing
x[!bad]##Logical is negated to get all the valid elements
```

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

- In the case of multiple things you want to take the subset of with no missing values

```
x <- c(1, 2, NA, 4, NA, 5, NA, 7)
y <- c("a", "b", NA, "d", NA, "f", "g", NA)
good <- complete.cases(x,y)
##Indicates which elements of either vectors are missing
good
```

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

```
##As such, final two elements print FLASE since there is an NA
##in at least one element
x[good]
```

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

```
y[good]
```

```
## [1] "a" "b" "d" "f"
```

```
airquality[1:6, ]## Returns first 6 rows
```

```
##   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)
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
```

```
##Returns first 6 rows that have don't have any missing values
```

- Additional note from `swirl()`

```
my_data <- sample(c(rnorm(100), rep(NA,100)), 20)
my_data
```

```
##  [1] -1.3353273 -1.1390477         NA         NA         NA         NA
##  [7]  0.6650119  1.2737941         NA  0.7848192         NA         NA
## [13]  0.7347085  1.7633301  1.6978597  0.6226662         NA  0.5568483
## [19]  0.3680870         NA
```

```
is.na(my_data)
```

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

```
## Returns a vector of logicals that indicate what positions of my_data are NA
my_data == NA
```

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

```
## Returns a vector of NAs because NA is a placeholder for a qty that's
##not available. Therefore the expression is incomplete and returns a
##vector of NAs the same length as my_data
```

Vectorized operations

- A feature of R that makes it easy to use on the command line
- Many operations in R are *vectorized* making code more efficient, concise, and easier to read.

```
x <- 1:4; y <- 6:9
x + y ##Adds vectors by position of elements
```

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

```
x >=2 ##Returns a logical vector that indicates which vectors are > or = 2
```

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

```
y == 8
```

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

```
x * y ##Multiplies each element of x by the respective element of y
```

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

```
x / y ##Divides by element
```

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

- Vectorized Matrix Operations

```
x <- matrix(1:4, 2, 2); y <- matrix(rep(10,4), 2, 2)
x * y ##element-wise multiplication
```

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

```
x / y
```

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

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

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

Stuff for quiz

```
##4
x <- 4L
class(x)#int?
```

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

```
##5
x <- c(4,TRUE)
class(x)
```

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

```
##6
x <- c(1,3,5)
y <- c(3,2,10)
rbind(x,y)
```

```
##   [,1] [,2] [,3]
## x    1    3    5
## y    3    2   10
```

```
##8
x <- list(2, "a", "b", TRUE)
class(x[[1]])
```

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

```
x[[1]]
```

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

```
##9
x <- 1:4
y <- 2:3
x+y
```

```
## [1] 3 5 5 7
```

```
class(x+y)
```

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

```
##10
x <- c(3, 5, 1, 10, 12, 6)
ans <- c(0, 0, 0, 10, 12, 6)
x[x %in% 1:5] <- 0
x
```

```
## [1]  0  0  0 10 12  6
```

```
data <- read.csv("hw1_data.csv")
##First 2 rows
```

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

```
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
```

```
##Num rows
nrow(data)
```

```
## [1] 153
```

```
#Extract final 2 rows
data[(nrow(data)-1):(nrow(data)),]
```

```
##     Ozone Solar.R Wind Temp Month Day
## 152    18     131  8.0   76     9  29
## 153    20     223 11.5   68     9  30
```

```
#Value of 47th row
data[47,]
```

```
##    Ozone Solar.R Wind Temp Month Day
## 47    21     191 14.9   77     6  16
```

```
#Number of missing Ozone
bad <-is.na(data[,1])
sum(bad)
```

```
## [1] 37
```

```
##Mean of ozone without NA
cleanData <- data[!bad,1]
mean(cleanData)##mean of Ozone, ignore NA
```

```
## [1] 42.12931
```

```
##Find mean of Solar.R where Ozone values are > 31 & Temp values are >90
bigOzone <- (data[,1]>31)
bigTemp <-(data[,4]>90&!is.na(data[,4]))
sOnBig <- data[bigOzone&bigTemp, 2]
bad <- is.na(sOnBig)
cleanSolar <- sOnBig[!bad]
mean(cleanSolar)
```

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

```
##What is the mean of "Temp" when "Month" is equal to 6
wheresSix <- (data[,5]==6)
sixMtemp <- data[wheresSix, 4]
mean(sixMtemp)
```

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

```r
##What was the maximum ozone value in month 5
wheresFive <- (data[,5]==5)
fivMonOz <- data[wheresFive, 1]
bad <- is.na(fivMonOz)
cleanFOzone <- fivMonOz[!bad]
max(cleanFOzone)
```

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

Misc Vanilla Functions

`dir.create()` - Creates a directory in current working directory (found with `getwd()`)
`args()` - Returns possible arguments of parameter
`file.<arguments>`
+ exists - checks if parameter exists, returns logical + info - returns info about file; such as: size, if it is a directory, mode, mtime, ctime, atime, uid, gid, username, groupname. `dir.create` - allows to manipulate directories and file permissions * *Sequence of Numbers* + The ':' operator

```r
1:20
```

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

```r
pi:10 ##Increments by 1 until number is > the upper limit, 10
```

```
## [1] 3.141593 4.141593 5.141593 6.141593 7.141593 8.141593 9.141593
```

```r
15:1 ##Decrementing is cool too
```

```
##  [1] 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1
```

- `seq()`

```r
seq(1,20) ##Equivelant to `1:20`
```

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

```r
seq(0,10,by=0.5)
```

```
##  [1]  0.0  0.5  1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5  5.0  5.5  6.0  6.5  7.0
## [16]  7.5  8.0  8.5  9.0  9.5 10.0
```

```r
my_seq <- seq(5,10,length=30)##sets `by` so the inc is consistent
seq_along(my_seq)#Creates a seq from 1:length(my_seq)
```

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

- `rep()` - creates a vector of a repeated value

```r
rep(0, times = 30)
```

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

```r
rep(c(0,1,2),times=10)##One can also use a vector as the argument
```

```
##  [1] 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2
```

```r
rep(c(0,1,2),each = 10)##Makes 10 of each
```

```
##  [1] 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2
```

- `paste()` - joins elements of a vector
  - `collapse` - argument that tells R what character to add inbetween each element.

```r
paste(1:3,c("X", "Y", "Z"), sep ="")#paste can also combine vectors
```

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

```r
paste(1:8,c("X", "Y", "Z"), sep ="")#even of diferent length
```

```
## [1] "1X" "2Y" "3Z" "4X" "5Y" "6Z" "7X" "8Y"
```

- `rnorm()` - draws from a standard normal distribution, number drawn is determined by parameter

```r
rnorm(10)
```

```
##  [1]  0.4876912 -1.3607416  1.7544849  1.5574902 -0.8534833 -1.4825295
##  [7]  0.0145354 -0.7289790 -0.2669828  0.1569245
```

- `sample()` - takes a sample of the specified size from the elements of x; `replace` is a logical argument that can be included

```r
sample(c(1:20),10)
```

```
##  [1] 18  1  9 10  4 11  8  2 14 20
```

```r
#`sample(c(1:5),10)`
##^would cause an error since replace defaults to false and n is bigger than N
sample(c(1:5),10,replace=TRUE)
```

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

- `identical()` - logical return; TRUE if the two objects are **exactly** equal

- A note on assigning names

```r
my_matrix <- matrix(1:20, 4, 5)##4x5 matrix
patients <- c("Bill", "Gina", "Kelly", "Sean")
cbind(patients, my_matrix)##NOughty!
```

```
##      patients
## [1,] "Bill"   "1" "5" "9"  "13" "17"
## [2,] "Gina"   "2" "6" "10" "14" "18"
## [3,] "Kelly"  "3" "7" "11" "15" "19"
## [4,] "Sean"   "4" "8" "12" "16" "20"
```

```r
my_data <- data.frame(patients, my_matrix)
my_data #The drake meme
```

```
##   patients X1 X2 X3 X4 X5
## 1     Bill  1  5  9 13 17
## 2     Gina  2  6 10 14 18
## 3    Kelly  3  7 11 15 19
## 4     Sean  4  8 12 16 20
```

Control structures, functions, scoping rules, dates and times

Learning Objectives

- Write an **if-else** expression
- Write a **for loop**, a **while loop**, and a **repeat loop**
- Define a function in R and specify its return value(see **Functions Part 1** and **Functions Part 2**)
- Describe **how R binds a value to a symbol via the search list**
- Define what **lexical scoping is** with respect to **how the value of free variables are resolved in R**
- Describe the difference between **lexical scoping and dynamic scoping rules**
- Convert a character string representing a date/time into an **R datetime object**.

Control Structures

- Control structures allow you to control the flow of execution of the program
  - `if, else`: testing a condition
  - `for`: execute a loop a fixed number of times
  - `while`:execute a loop *while* a condition is true
  - `repeat`: execute an infinite loop
  - `break`: break the execution of a loop
  - `next`: skip an interation of a loop
  - `return`: exit a function

- Infinite loops should generally be avoided, even if they are theoretically correct
- For command-line interacte work, the *apply functions are more useful

if-else

- syntax can be just like `cpp`

```
x <- sample(1:6, 1)
if(x>3){
  y <- 10
} else {
  y <- 0
}
vals <- c(x,y)
vals
```

```
## [1] 3 0
```

- but `R` also excepts other syntax

```
x <- sample(1:6, 1)
y <- if(x>3){
  10
} else {
  0
}
vals <- c(x,y)
vals
```

```
## [1] 3 0
```

- nested `if`s and "else-less" `if`s are also acceptable

for loop

- `for` loops take an interator variable and assign it to successive values from a sequence or vector.
  - Common for iterating over the elements of an object

```
for(i in 1:10){
  print(i)
}
```

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

```
## [1] 7
## [1] 8
## [1] 9
## [1] 10
```

- R is flexible in how you can index different objects; the following loops are all equivelent

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

for(i in 1:4){
  print(x[i])
}
```

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

```
for (i in seq_along(x)) {##seq_along creates an integer sequence that's as long as x
  print(x[i])
}
```

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

```
for(letter in x) {##letter is assigned to the nth element of x
  print(letter)
}
```

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

```
for(i in 1:4) print(x[i])##{} can be omitted for a single element in the body
```

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

- Nested for loops are also acceptable

while loop

```
count <- 0
while(count < 10){
  print(count)
```

```
  count <- count + 1##Make sure you increment
}
```

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

*A while loop with logical operators

```
z <- 5
while(z>=3 & z <= 10){
  print(z)
  coin <- rbinom(1,1,0.5)
  if(coin == 1){ ##random walk
    z <- z+1
  } else {
      z <- z-1
    }
}
```

```
## [1] 5
## [1] 6
## [1] 5
## [1] 6
## [1] 7
## [1] 6
## [1] 5
## [1] 6
## [1] 7
## [1] 6
## [1] 5
## [1] 4
## [1] 5
## [1] 6
## [1] 5
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 6
## [1] 5
## [1] 4
```

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

- Conditionals will "**short circuit**" when evaluating `&` or `|`

Repeat, Next, Break

- Not common in statistical applications
- only way to exit a `repeat` loop is to call `break`

```
x0 <- 1
tol <- 1e-8
count <- 0

repeat{
  count <- count + 1
  x1 <- sample(seq(-1,1,length.out = 1000), 1)

  if(abs(x1 - x0) < tol){
    outVect <- c("In ", count," loops we found a x0, ", x0, ", that was within ", tol, " of x1
    output <- paste(outVect, collapse = "")
    print(output)
    break
  } else {
    x0 <- x1
  }
}
```

```
## [1] "In 401 loops we found a x0, 0.235235235235235, that was within 1e-08 of x1, 0.23523523
```

- loops that are not guaranteed to stop ought to have a hard limit on the number of iterations(e.g. using a for loop instead) and then report whether convergence was achieved or not

*`next` - used to skip an iteration of a loop

```
for(i in 1:100){
  if(i <= 20){
    ##Skip the first 20 iterations
    next
  }
  ## Other code could go here
}
```

Functions

Your first R function(s)

```
add2 <- function(x, y) {
  x + y
}

add2(3,5)
```

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

```
above10 <- function(x){
  use <- x > 10
  x[use]
}

above <- function(x, n = 10){##n value is defaulted to 10
  use <- x > n
  x[use]
}

y <- 1:20
above10(y)
```

```
##  [1] 11 12 13 14 15 16 17 18 19 20
```

```
above(y, 12)
```

```
## [1] 13 14 15 16 17 18 19 20
```

```
columnmean <- function(y, removeNA=TRUE) {
  nc <- ncol(y)
  means <- numeric(nc)##numeric vector, length same as value of nc
  for(i in 1:nc) {
    means[i] <-  mean(y[,i], na.rm = removeNA)##many functions have the option to remove NAs
  }
  means
}

columnmean(airquality)
```

```
## [1]  42.129310 185.931507   9.957516  77.882353   6.993464  15.803922
```

```
columnmean(airquality, FALSE)
```

```
## [1]        NA        NA  9.957516 77.882353  6.993464 15.803922
```

Functions pt 1

- created using the `function()` directive

- stored as R objects, of the class "function".
- Function in R are "first class objects", as such..
  - they can be passed as arguments to other functions

  - they can be nested, so that you can define a function inside of another function.
- The return value of a function is the last expression in the function body to be evaluated

- Functions have **named arguments** whch potentially have **default values**
  - The **formal arguments** are the arguments included in the function definition
  - The `formals` function returns a list of all the formal arguments of a function
  - Not every function call in R makes use of all the formal arguments, they can be missing or have a default set.

- R function arguments can be matched positionally or by name.
  - So all the following calls to `sd` are equivalent

```
mydata <- rnorm(100)
sd(mydata)
```

```
## [1] 0.9931813
```

```
sd(x = mydata)
```

```
## [1] 0.9931813
```

```
sd(x = mydata, na.rm = FALSE)
```

```
## [1] 0.9931813
```

```
sd(na.rm = FALSE, x = mydata)
```

```
## [1] 0.9931813
```

```
sd(na.rm = FALSE, mydata)## when picking a position R fills to earliest, undeclared argument
```

```
## [1] 0.9931813
```

- Messing around with the order of the arguments can lead to confusion, as such it's not recommended

- Positional matching and matching by name can be mixed, which is helpful when functions have many arguments and one doesn't need them all, Ex:

```
args(lm)
```

```
## function (formula, data, subset, weights, na.action, method = "qr",
##     model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE,
##     contrasts = NULL, offset, ...)
## NULL
```

- The following two calls are equivalent
  - `lm(data = mydata, y - x, model = FALSE, subset = 1:100)`
  - `lm( y~x, mydata, 1:100, model = FALSE, x=TRUE)`
- Second option is ideal way to mix and match since first 3 are gonna be specified by position
- Function arguments can also be **partially** matched by the following Order of operations:
  1. Check for exact match for a named argument
  2. Check for a partial match
  3. Check for a positional match

Functions pt 2

- In addition to not specifying a defauly value, you can also set an argument value to `NULL`

```
f <-function(a, b = 1, c = 2, d = NULL) {
  ##<super_rad_code.txt>
}
```

- Arguments to functions are evaluated **lazily**, so they are evaluated only as needed

```
f <- function(a,b) {
  a^2
}
f(2) ##b is not evaluated in the function, so an error does not occur
```

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

```
f <- function(a,b) {
  print(a)
  print(b)
}
tryCatch(f(45), error=function(e){print("Error in print(b) : argument \"b\" is missing, with no
```

```
## [1] 45
## [1] "Error in print(b) : argument \"b\" is missing, with no default"
```

```
  ##The function is evaluated until print(b) tries to execute
```

- The "..." Argument
  - ... indicate a variable number of arguments that are usually passed on to other functions.
  - Often used when extending another function and you don't want to copy the entire argument list of the og function

```
myplot <- function(x,y, type = "1", ...) {
  plot(x,y,type = type, ...)
}
```

- Also used with Generic function so that extra arguments can be passed to methods (more on this later)

```
mean
```

```
## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x559ea70241e0>
## <environment: namespace:base>
```

- The `...` argument is also necessary when the number of arguments passed to the fucntion cannot be known in advance

```
args(paste)
```

```
## function (..., sep = " ", collapse = NULL)
## NULL
```

```
args(cat)
```

```
## function (..., file = "", sep = " ", fill = FALSE, labels = NULL,
##     append = FALSE)
## NULL
```

- Arguments that appear **after** `...` on the arguemnt list must be named explicitly and cannot be partially matched

```
paste( "a", "b", sep = ":")
```

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

```
paste( "a", "b", se = ":")##Attempt to partially match ‘sep‘
```

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

Scoping Rules

Symbol Binding

- How does R decide what value to assign to a symbol in it's body?

```
lm <- function(x) {x*x} ##lm is already a function in the ‘stats‘ package
lm
```

```
## function(x) {x*x}
```

- R searches through diffrent environments when attempting to bind a value to a symbol

1. The global environment - which is your workspace in the Evironment pane

- iow, local variables are prioritized

2. Namepaces of each of the packages- in order of the search list

- `search()` displays the search list

```
search()##.GlobalEnv is always [1]
```

```
## [1] ".GlobalEnv"        "package:stats"     "package:graphics"
## [4] "package:grDevices" "package:utils"     "package:datasets"
## [7] "package:methods"   "Autoloads"         "package:base"
```

- *base* package is always last in the search list
- User's can configure which packages get loaded, as such one cannot assume a set list of packages will be available.
  - When a user loads a package with `library` the namespace gets put in `[2]` of the search (list) stack.

- Note that R has separate namespaces for functions and non-functions
  - So it's possible to have an object named `c` and a function named `c`

  - However only once symbol can be named `c` in your `.GlobalEnv`
- Scoping rules for R are the main feature that make it different from the original `S` language
  - R uses *lexical scoping* or *static scoping*. A common alternative is *dynamic scoping*
    * lexical scoping turns out to be particularly useful for simplifying statistical computations

  - These rules determine how a value is associated with a free variable in a function

Free Variables

- Consider the following,

```
f <- function(x,y) {
  x^2 + y / z
}
```

- This function has 2 formal arguments `x` and `y`. The additional symbol, `z`, is called a *free variable*.

- A **free variable** is neither a formal argument nor a local variable

Lexical Scoping

- *the values of free variables are searched for in the environment in which the function was defined*

- What is an environment

  - An *environment* is a collection of (symbol, value) pairs, i.e. `x` is a symbol and `3.14` might be its value.

  - Every environment has a parent environment; thus it is possible for an environment to have multiple "children"

  - the only environment without a parent is the *empty environment*
  - **a closure* or *function closure** - A function that is associated with an environment
    * Key to a lot of interesting operations in `R`

- Searching for the value for a free variable

  - Search starts in the environment in which a function was defined

  - Then the search is dontinued in the *parent environment*

  - The search continues down the sequence of parent environments until..
  - We hit the *top-level environment*; this is usually the global environment (workspace) or the namespace of a package.

  - After the top-level env. the search continues down the search list until..
  - We hit the *empty environment*, at which point an error is thrown.

- Other languages that support lexical scoping:

  - Scheme

  - Perl

  - Python

  - Common Lisp (**all languages converge to Lisp**)

Scoping Rules (sub)

- R's big sellin' point is that you can functions defined *inside other functions*
  - Languages like `C` don't let you do this

  - In this case, the environment in which a function is defined is the body of another function
  - As such, a function can return a function

```
make.power <-  function(n) {
  pow <-  function(x) {
    x^n
  }
  pow
}
## This function returns another function as is value
```

```
## Which allows us to create functions as follows
cube <- make.power(3)
square <- make.power(2)
cube(3)
```

```
## [1] 27
```

```
square(3)
```

```
## [1] 9
```

- How do you know what a function's environment is?

```
ls(environment(cube)) ##Returns objects within environment that a function was defined
```

```
## [1] "n"   "pow"
```

```
get("n", environment(cube)) ##Returns the value of "n" within the 'cube' environment
```

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

```
ls(environment(square))
```

```
## [1] "n"   "pow"
```

```
get("n", environment(square))
```

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

- Example of ***dynamic scoping***

```
y <- 10

f <- function(x) {
  y <- 2##assign new value to 2 within function's scope
  y^2 + g(x) ##2^2 + g(x)
}

g <- function(x) {
  x*y ## computes x*10, regardless if y got reassigned elsewhere
}

f(3) ## Computes (2^2 + 2*10) due to dynamic scoping that is simulated here
```

```
## [1] 34
```

- When a in the global environment and is subsequently *called* from the global enironment, then the defining environment and the calling environment are the same. This can give the illusion of dynamic scoping.

```
g <- function(x) {
  a <- 3
  x+a+y
}
```

```
y <- 3
g(2)
```

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

- This occurs because GlobalEnv is always first in the search list, as such g is in the same environment as when we call it and `y` exists

###Consequences of Lexical Scoping * In R, all objects must be stored in memory
* All functions must carry a pointer to their respective defining environment, which could be anywhere
* In S-PLUS, free variables are always looked up in the global workspace, so everything can be stored on the disk because the "defining environment" of all functions is the same.

Aside: Likelihood & Log-likelihood

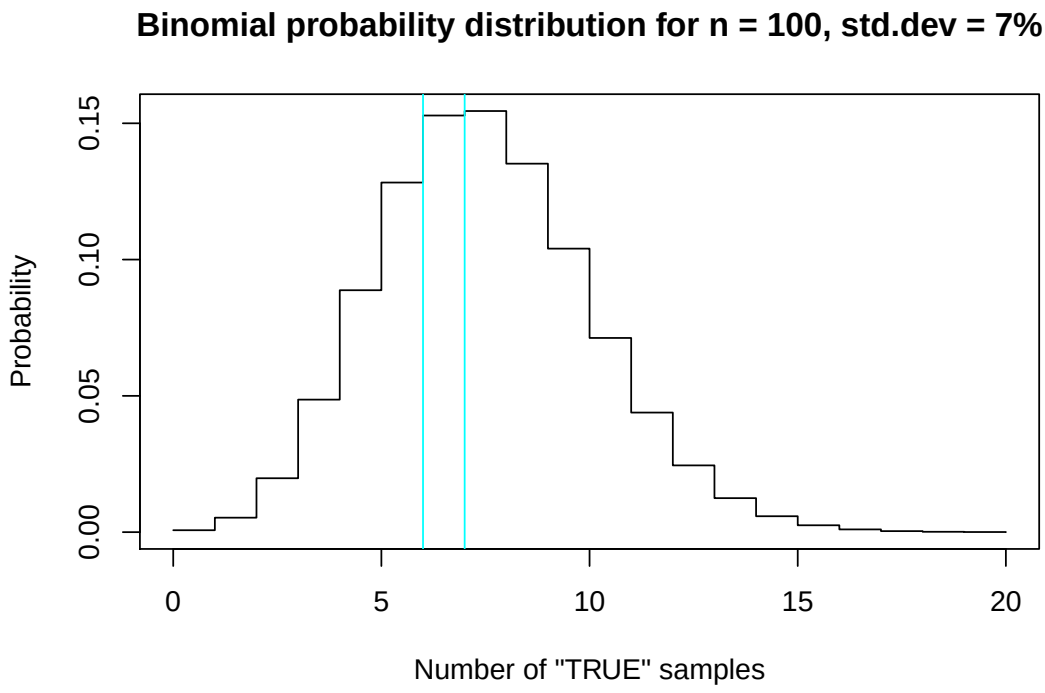
- **Notes are from this video**

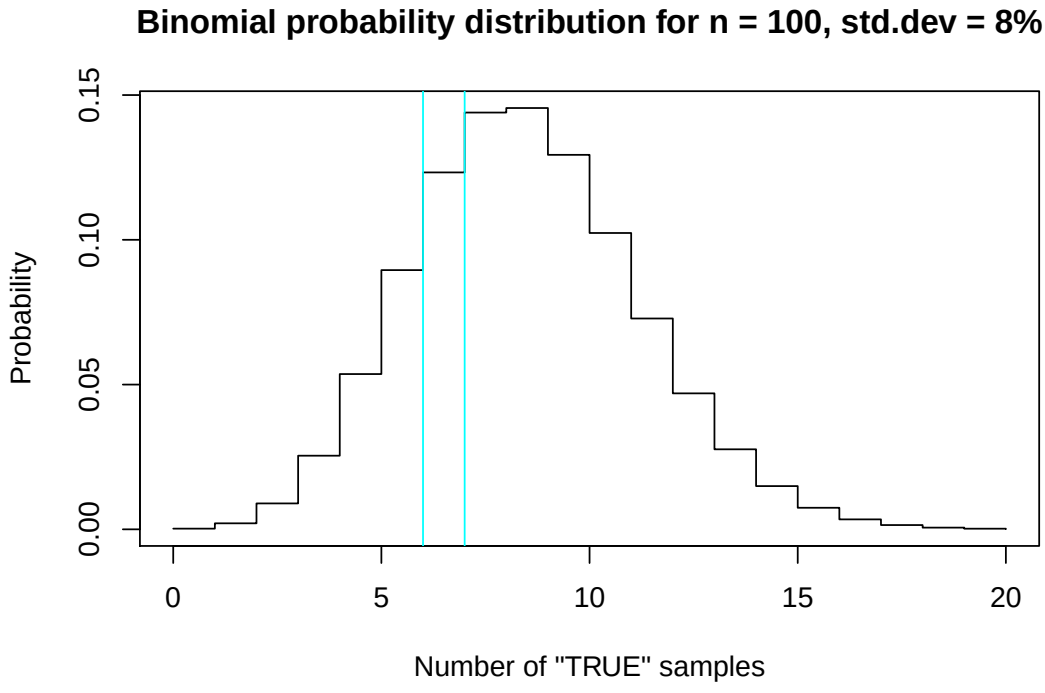
- Probability review
  - proportion of times a given outcome would occur in many trials.
    - Iow, It is the long-term relative frequency

  - P(weight between 32 and 34 grams | mean = 32 and std. deviation = 2.5)
  - Read as: "Probability of a weight between 32 and 34 grams, given that the mean is 32 and the standard deviation is 2.5"

- Likelihood
  - *likelihood* - describes the extent to which the sample provides support for any particular parameter value. Higher support corresponds to a higher value for the likelihood.

  - Likelihood describes probability of distribution factors, such as mean or std. deviation





– If a sample of 100 was taken and 6 people returned TRUE, which of the above models is more ***Likely***?

– Likelihood of either model is the probability at 6.

  - A **likelihood function** provides a model for a fixed sample outcome.
  - Individual likelihood values are *meaningless*
  - Comparing two values, however, is *informative*
  - As such we ussually discuss a **Likelihood Ratio** - L(theta[1];y) / L(theta[2];y)
    * Suppose the following:

```
L_Of_Theta_0.07 <- 0.152
L_Of_Theta_0.08 <- 0.123
Liklihood_Ratio <- function(a, b) {
  round(a/b, digits = 3)
}
Liklihood_Ratio(L_Of_Theta_0.07, L_Of_Theta_0.08)
```

```
## [1] 1.236
```

```
## [1] "Thus, a population prevalence of 7% has 1.236 times the support of"
## [2] "a population prevalence of 8% (given our sample)"
```

- **likelihood function** - for a given sample, it creates the likelihoods for all possible values of theta
- In summary, in ***likelihood functions***, *the mean* or *the standard deviation* is the parameter

Optimization

- Optimization routines in R, like `optim`, `nlm`, and `optimize` require you to pass a function, whose argument is a vector of parameters (e.g. a **log-likelihood**)
- Optimization tries to find the *min* or *max* of a given function
  - An object function might depend on a host of other things besides its parameters (like *data*)
- When writing software which does optimization, it may be desirable to allow the user to hold certain parameters fixed
- *Note*: Optimization functions in R *minimize* functions, so you need to use the negative log-likelihood

```
make.NegLogLik <- function(data, fixed=c(FALSE,FALSE)) {
  params <- fixed
  function(p) {
    params[!fixed] <- p #Assigns p to whatever variable wasn't fixed
    mu <- params[1]
    sigma <- params[2]
    a <- -0.5*length(data)*log(2*pi*sigma^2)
    b <- -0.5*sum((data-mu)^2) / (sigma^2)
    -(a+b)
```

```
  }
}

set.seed(1); normals <- rnorm(100, 1, 2)
nLL <- make.NegLogLik(normals)
nLL
```

```
## function(p) {
##     params[!fixed] <- p #Assigns p to whatever variable wasn't fixed
##     mu <- params[1]
##     sigma <- params[2]
##     a <- -0.5*length(data)*log(2*pi*sigma^2)
##     b <- -0.5*sum((data-mu)^2) / (sigma^2)
##     -(a+b)
##   }
## <bytecode: 0x559ea821c908>
## <environment: 0x559ea6fa57d0>
```

```
#<environment: ...> tells you the address of the pointer to the defining environment
ls(environment(nLL))##Lists Values in function's environment
```

```
## [1] "data"   "fixed"  "params"
```

```
optim(c(mu=0, sigma = 1), nLL)$par
```

```
##       mu    sigma
## 1.218239 1.787343
```

```
##Returns estimates for mu&sigma by minimizing neg. liklihood

nLL <- make.NegLogLik(normals, c(FALSE, 2))##Fixing sigma to 2
optimize(nLL, c(-1, 3))$minimum
```

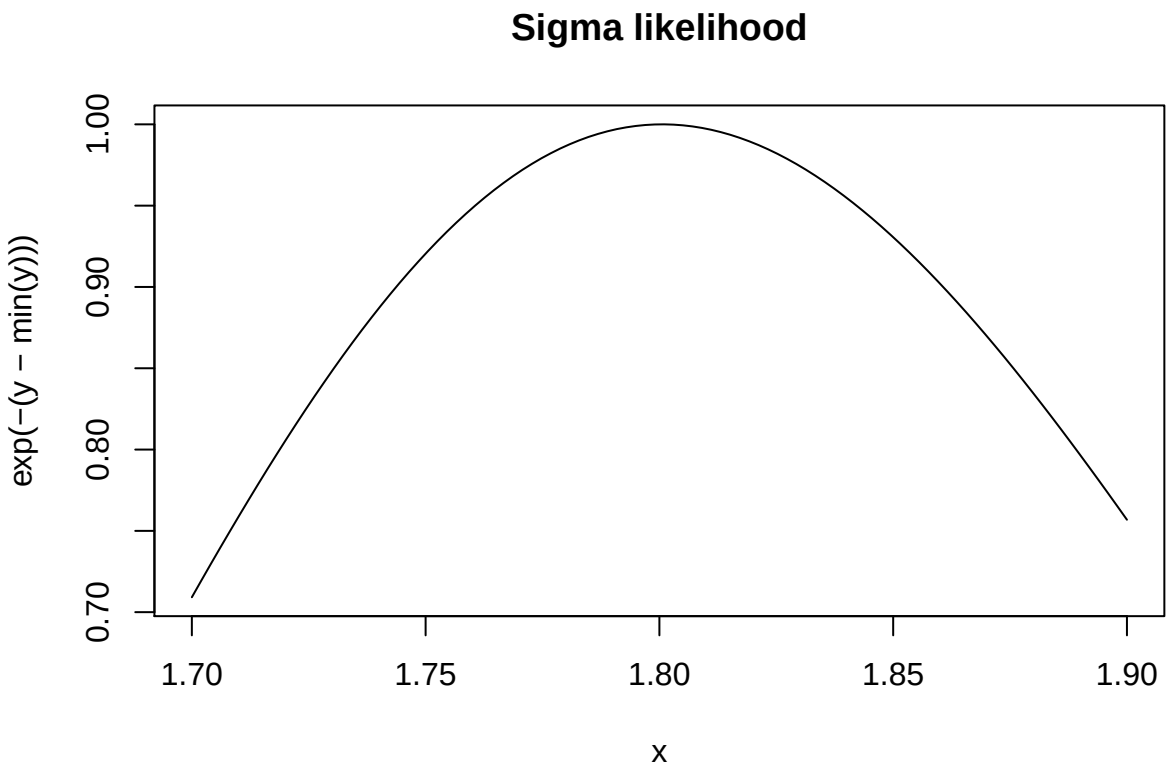
```
## [1] 1.217775
```

```
nLL <- make.NegLogLik(normals, c(1, FALSE))##Fixing mu to 1
optimize(nLL, c(1e-6, 10))$minimum
```

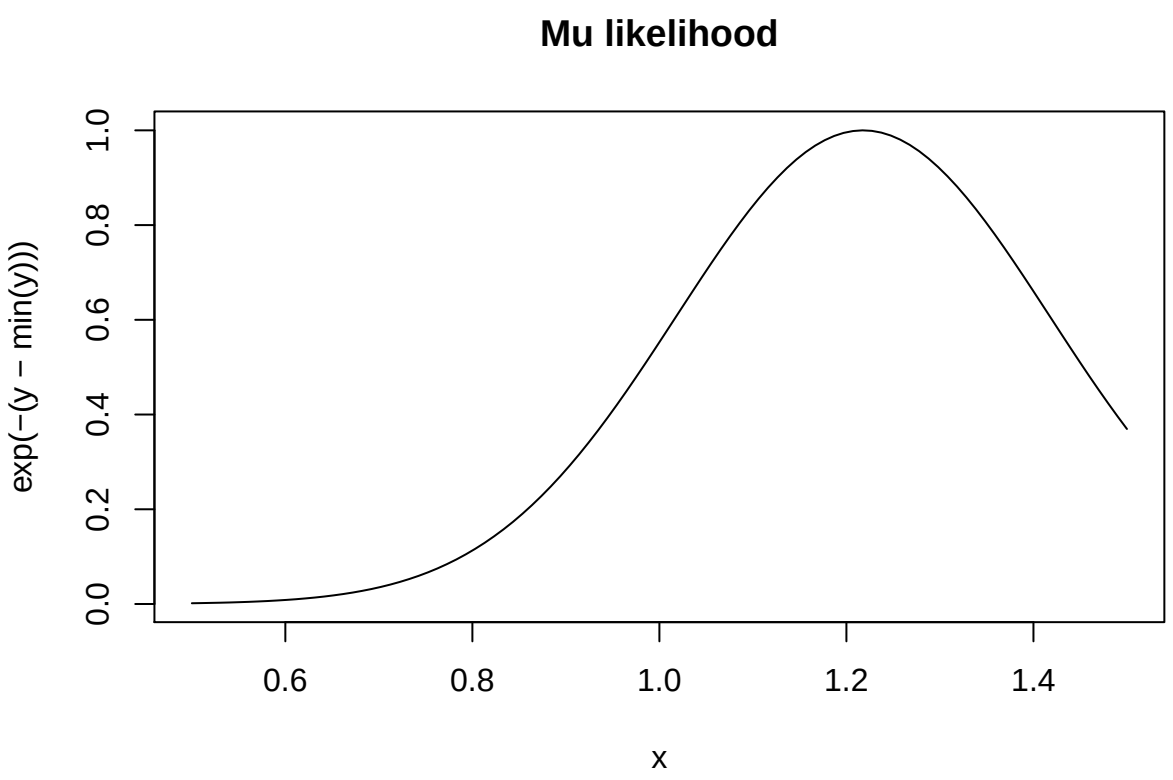
```
## [1] 1.800596
```

- `optimize` will only optimize a function with a single missing variable
  - `optim` can optimize a function with more than one missing variable

```
##The following plots the sigma likelihood
nLL <- make.NegLogLik(normals, c(1, FALSE))
x <- seq(1.7, 1.9, len=100)
y <- sapply(x, nLL)
plot(x, exp(-(y - min(y))), type = "l", main = "Sigma likelihood")
```




```
##The following plots the mu likelihood
nLL <- make.NegLogLik(normals, c(FALSE, 2))
x <- seq(0.5, 1.5, len = 100)
y <- sapply(x, nLL)
plot(x, exp(-(y - min(y))), type = "l", main = "Mu likelihood")
```



- Objective function can be "built" which contain all of the necessary data for evaluating the function
- No need to carry around long argument lists
  - useful for interactive and exploratory work
- Code can be simplified and cleaned up

Coding Standards

- "Help make your code readable. . . just like it is with any other *style*, like your clothing, it's hard to get everyone to agree on one set of ideas. But there are some basic/minimal standards for coding `R`"

1. Code should always be written with a text editor and saved as `.txt`

- Can be read by any basic editting program; makes code versatile
- RStudio saves code as text by default

2. Indent your code

3. Limit the width of your code (80 columns)

- indents and column width can be editted in settings of RStudio
- these limits help promote clean code

4. Limit the length of individual functions

- Each function ought to only do one activity
  - `readTheData` should read and return the data, NOT read the data, process it, fit a model, and then print an output.
- Nice to read an entire function that is able to fit on one screen of the editor
- Helps when debugging

Dates and Times

- Dates are represented by the `Date` class
  - Stored internally as the number of days since 1970-01-01
  - Can be coerced from a character string using the `as.Date()` function
  - When printed they default to converting back to a readable character string
  - If `unclass`ed, they'll display the numeric value R is storing them as

```
x <- as.Date("1970-01-01")
x
```

```
## [1] "1970-01-01"
```

```
unclass(x)
```

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

```
unclass(as.Date("1970-01-02"))##One day since origin
```

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

- Times are represented by the `POSIXct` or the `POSIXlt` class
  - Stored internally as the number of seconds since 1970-01-01
    - Negative times are also valid
  - `POSIXct` is just a very large integer; useful when one wants to store times in something like a data frame
    - `c` for "concise"

```
x <- Sys.time()
x ## Already in 'POSIXct' format
```

```
## [1] "2020-01-29 05:57:55 EST"
```

```
unclass(x)
```

```
## [1] 1580295476
```

```
#x$sec
##Throws: "Error in x$sec : $ operator is invalid for atomic vectors"
p <- as.POSIXlt(x) ##After this conversion sec can be extracted
p$sec
```

```
## [1] 55.6215
```

```
str(p)
```

```
##  POSIXlt[1:1], format: "2020-01-29 05:57:55"
```

- `POSIXlt` is a list "underneath" and it stores meta data such as: the day of the week, day of the year, month, day of the month
  - `l` for "list"

```
x <- Sys.time()
x
```

```
## [1] "2020-01-29 05:57:55 EST"
```

```
p <- as.POSIXlt(x)
names(unclass(p))
```

```
##  [1] "sec"    "min"    "hour"   "mday"   "mon"    "year"   "wday"   "yday"
##  [9] "isdst"  "zone"   "gmtoff"
```

```
p$sec
```

```
## [1] 55.6372
```

- `strptime` function - lets you convert dates written in different formats into `POSIXlt`
  - Check `?strptime` for details on the string formatting

```
datestring <- c("January 10, 2012 10:40", "December 9, 2011 9:10")
x <- strptime(datestring, "%B %d, %Y %H:%M")
x
```

```
## [1] "2012-01-10 10:40:00 EST" "2011-12-09 09:10:00 EST"
```

```
class(x)
```

```
## [1] "POSIXlt" "POSIXt"
```

*Operations on Dates and Times + Some mathematical operations work (`+`, `-`, logicals (i.e. `==`, `<=`)) + You can't mix classes

```
x <- as.Date("2012-01-01")
y <- strptime("9 Jan 2011 11:34:21", "%d %b %Y %H:%M:%S")
#x-y ##Throws:
## Incompatible methods ("-.Date", "-.POSIXt") for "-"
## Error: non-numeric argument to binary operator
x <- as.POSIXlt(x)
x-y
```

```
## Time difference of 356.3095 days
```

- Some generic functions that work on dates and times
  - `weekdays`: returns the day of the week
  - `months`: returns the month name
  - `quarters`: returns the quarter number ("Q1", "Q2", "Q3", or "Q4")
- Dates and Times classes keep track of *leap years*, *leap seconds*, *daylight savings*, and *time zones

```
x <- as.Date("2012-03-01")
y <- as.Date("2012-02-28")
x-y
```

```
## Time difference of 2 days
```

```
x <- as.POSIXct("2012-10-25 01:00:00")
y <- as.POSIXct("2012-10-25 06:00:00", tz = "GMT")
y-x
```

```
## Time difference of 1 hours
```

Datetime Object

- Summary
  - Dates and times have special classes in R that allow for numerical and statistical calculations
  - Dates use the `Date` class
  - Times use the `POSIXct` and `POSIXlt` class
  - Character strings can be coerced to date/Time classes using the `strptime`, `as.Date`, `as.POSIXlt`, or `as.POSIXct` functions.
  - A lot of plotting functions will recognize Datetime objects

Misc Vanilla Functions

- `rm(list=ls())` - clears everything from workspace
- `&&` operator will only evaluate the first element of a vector and return a single logical; whereas `&` will evaluate all elements and return a logical vector
  - likewise for the `||` and `|` operators
- All `&` operators are evaluated before `|` operators

- `xor()` evaluates arguments with *exclusive or*
- `which()` returns a vector that indicates which indicies are `TRUE`
- `any()` returns true if any element is true in the logical vector passed as an argument
- `all()` returns true if all the elements in the logical vector passed as an argument are `TRUE`
- *Note*: John Chambers, the creator of R once said: "*To understand computations in R, two slogans are helpful:* 1. *Everything that exists is an object.* 2. *Everything that happens is a function call.*"
- This is a strict rule in R programming: all arguments after an ellipses must have default values.
- Let's say I wanted to define a binary operator that multiplied two numbers and then added one to the product. Notice the `%` and `"` surrounding the operator name. An implementation of that operator is below:

```
"%mult_add_one%" <- function(left, right){ # Notice the quotation marks!
  left * right + 1
}

4 %mult_add_one% 5
```

```
## [1] 21
```

Loop functions, debugging tools

Simulation, code profiling