**R Programing**

Week 1 - John Hopkins Coursera

***L1 - Overview and History of R***

**What is S?**

* S is a language that was developed by John Chambers and others at Bell Labs.
* S was initiated in 1976 as an internal statistical analysis environment—originally implemented as Fortran libraries.
* Early versions of the language did not contain functions for statistical modeling.
* In 1988 the system was rewritten in C and began to resemble the system that we have today (this was Version 3 of the language). The book Statistical Models in S by Chambers and Hastie (the white book) documents the statistical analysis functionality.

**Features of R**

* Syntax is very similar to S, making it easy for S-PLUS users to switch over.
* Semantics are superficially similar to S, but in reality are quite different (more on that later).
* Runs on almost any standard computing platform/OS (even on the PlayStation 3)
* Frequent releases (annual + bugfix releases); active development.
* Quite lean, as far as software goes; functionality is divided into modular packages
* Graphics capabilities very sophisticated and better than most stat packages.
* Useful for interactive work, but contains a powerful programming language for developing new tools (user -> programmer)
* Very active and vibrant user community; R-help and R-devel mailing lists and Stack Overflow
* It's free! (Both in the sense of beer and in the sense of speech.)

**Free Software**

* With free software, you are granted
* The freedom to run the program, for any purpose (freedom 0).
* The freedom to study how the program works, and adapt it to your needs (freedom 1). Access to the source code is a precondition for this.
* The freedom to redistribute copies so you can help your neighbor (freedom 2).
* The freedom to improve the program, and release your improvements to the public, so that the whole community benefits (freedom 3). Access to the source code is a precondition for this.

**Drawbacks of R**

* Essentially based on 40 year old technology.
* Little built in support for dynamic or 3-D graphics (but things have improved greatly since the “old days”).
* Functionality is based on consumer demand and user contributions. If no one feels like
* implementing your favorite method, then it’s your job!
* Objects must generally be stored in physical memory; but there have been advancements to deal with this too
* Not ideal for all possible situations (but this is a drawback of all software packages).

Swirl notes about Workspace and Files:

| Determine which directory your R session is using as its current working directory using

| getwd().

> getwd()

[1] "C:/Users/Nuno.Rodrigues/Documents"

| List all the objects in your local workspace using ls().

> ls()

[1] "my\_div" "my\_sqrt" "x" "y" "z"

| List all the files in your working directory using list.files() or dir().

> dir()

[1] "Access Developer Extensions"

[2] "Adobe"

….

| Use the args() function to determine the arguments to list.files().

> args(list.files)

function (path = ".", pattern = NULL, all.files = FALSE, full.names = FALSE,

recursive = FALSE, ignore.case = FALSE, include.dirs = FALSE,

no.. = FALSE)

NULL

| Create a file in your working directory called "mytest.R" using the file.create() function.

> file.create("mytest.R")

[1] TRUE

| Check to see if "mytest.R" exists in the working directory using the file.exists()

| function.

> file.exists("mytest.R")

[1] TRUE

| Access information about the file "mytest.R" by using file.info().

> file.info("mytest.R")

size isdir mode mtime ctime atime exe

mytest.R 0 FALSE 666 2015-10-14 19:08:43 2015-10-14 19:08:43 2015-10-14 19:08:43 no

| Create a directory in the current working directory called "testdir2" and a subdirectory

| for it called "testdir3", all in one command by using dir.create() and file.path().

> dir.create(file.path('testdir2', 'testdir3'), recursive = TRUE)

***L4 to 11 –Data Types***

***Objects***

R has five basic or “atomic” classes of **objects**:

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

The most basic object is a **vector**:

A vector can only contain objects of the same class

BUT: The one exception is a **list**, which is represented as a vector but can contain objects of

different classes (indeed, that’s usually why we use them)

Empty vectors can be created with the vector() function.

***Numbers***

Numbers in R a generally treated as numeric objects (i.e. double precision real numbers)

If you explicitly want an integer, you need to specify the **L** suffix

Ex: Entering 1 gives you a numeric object; entering 1L explicitly gives you an integer.

There is also a special number **Inf** which represents infinity; e.g. 1 / 0; Inf can be used in

ordinary calculations; e.g. 1 / Inf is 0

The value **NaN** represents an undefined value (“not a number”); e.g. 0 / 0; **NaN** can also be

thought of as a missing value (more on that later)

***Attributes***

R objects can have attributes

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

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

***Vectors and Lists***

The **c()** function 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 <- 9:29 ## integer

> x <- c(1+0i, 2+4i) ## complex

Using the **vector**() function

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

> x

[1] 0 0 0 0 0 0 0 0 0 0

Swirl Notes:

**Vector** is a small collection of numbers. Numeric vectors can be used in arithmetic expressions => z \* 2 + 100 is [1] 102.20 118.00 106.28

**c() function**, which stands for 'concatenate' or 'combine';

| When given two vectors of the same length, R simply performs the specified arithmetic

| operation (`+`, `-`, `\*`, etc.) element-by-element. If the vectors are of different

| lengths, R 'recycles' the shorter vector until it is the same length as the longer vector.

| When we did z \* 2 + 100 in our earlier example, z was a vector of length 3, but technically

| 2 and 100 are each vectors of length 1.

| Behind the scenes, R is 'recycling' the 2 to make a vector of 2s and the 100 to make a

| vector of 100s. In other words, when you ask R to compute z \* 2 + 100, what it really

| computes is this: z \* c(2, 2, 2) + c(100, 100, 100).

**Mixing Objects**

What about the following?

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

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

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

When diferent objects are mixed in a vector, **coercion occurs so that every element in the vector is of the same class.**

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

> x <- 0:6

> class(x)

[1] "integer"

> as.numeric(x)

[1] 0 1 2 3 4 5 6

> as.logical(x)

[1] FALSE TRUE TRUE TRUE TRUE TRUE TRUE

> as.character(x)

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

Nonsensical coercion results in NAs:

> x <- c("a", "b", "c")

> as.numeric(x)

[1] NA NA NA

Warning message:

NAs introduced by coercion

> as.logical(x)

[1] NA NA NA

> as.complex(x)

[1] NA NA NA

Warning message:

NAs introduced by coercion

**Lists**

Lists are a special type of vector that **can contain elements of different classes**. Lists are a very

important data type in R and you should get to know them well.

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

> x

[[1]]

[1] 1

[[2]]

[1] "a"

[[3]]

[1] TRUE

[[4]]

[1] 1+4i

**Matrices**

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

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

> m

[,1] [,2] [,3]

[1,] NA NA NA

[2,] NA NA NA

> dim(m)

[1] 2 3

> attributes(m)

$dim

[1] 2 3

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

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

> m

[,1] [,2] [,3]

[1,] 1 3 5

[2,] 2 4 6

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

> m <- 1:10

> m

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

> dim(m) <- c(2, 5)

> m

[,1] [,2] [,3] [,4] [,5]

[1,] 1 3 5 7 9

[2,] 2 4 6 8 10

14

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

> 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**

Factors are used to represent categorical data. Factors can be unordered or ordered. One can think of a factor as an integer vector where each integer has a label.

* Factors are treated specially by modelling functions like **lm()** and **glm()**
* Using factors with labels is better than using integers because factors are self-describing; having a variable that has values “Male” and “Female” is better than a variable that has values 1 and 2.

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

> x

[1] yes yes no yes no

Levels: no yes

> table(x)

x

no yes

2 3

> unclass(x)

[1] 2 2 1 2 1

attr(,"levels")

[1] "no" "yes"

The order of the levels can be set using the levels argument to **factor()**. This can be important

in linear modelling because the first level is used as the baseline level.

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

levels = c("yes", "no"))

> x

[1] yes yes no yes no

Levels: yes no

**Missing Values**

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

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

> 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**

Data frames are used to store tabular data

* They are represented as a special type of list where every element of the list has to have the same length
* Each element of the list can be thought of as a column and the length of each element of the list is the number of rows
* Unlike matrices, data frames can store different classes of objects in each column (just like lists); matrices must have every element be the same class
* Data frames also have a special attribute called **row.names**
* Data frames are usually created by calling **read.table()** or **read.csv()**
* Can be converted to a matrix by calling **data.matrix() \***in this situation R will coercion all data to the same data type, it can result in something unexpected.

> x <- **data.frame**(foo = 1:4, bar = c(T, T, F, F))

> x

foo bar

1 1 TRUE

2 2 TRUE

3 3 FALSE

4 4 FALSE

> nrow(x)

[1] 4

> ncol(x)

[1] 2

**Names**

R objects can also have names, which is very useful for writing readable code and self-describing objects.

> x <- 1:3

> names(x)

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)

> x

$a

[1] 1

$b

[1] 2

$c

[1] 3

And matrices.

> m <- matrix(1:4, nrow = 2, ncol = 2)

> dimnames(m) <- list(c("a", "b"), c("c", "d"))

> m

c d

a 1 3

b 2 4

***L12 to 16 – Reading***

**Reading Tabular Data**

There are a few principal functions reading data into R.

* read.table, read.csv, for reading tabular data => 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

**Writing Data**

There are analogous functions for writing data to files

* write.table
* writeLines
* dump
* dput
* save
* serialize

There are analogous functions for writing data to files

* write.table
* writeLines
* dump
* dput
* save
* serialize

The **read.table** function is one of the most commonly used functions for reading data.

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

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

R will automatically

* skip lines that begin with a #
* figure out how many rows there are (and how much memory needs to be allocated)
* figure what type of variable is in each column of the table Telling R all these things directly makes
* R run faster and more efficiently.
* read.csv is identical to read.table except that the default separator is a comma.

**Reading in Larger Datasets with read.table**

With much larger datasets, doing the following things will make your life easier and will prevent R from choking.

* Read the help page for read.table, which contains many hints
* Make a rough calculation of the memory required to store your dataset. If the dataset is larger
* than the amount of RAM on your computer, you can probably stop right here.
* Set comment.char = "" if there are no commented lines in your file.

Use the colClasses argument. Specifying this option instead of using the default can make

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

each column is the following:

initial <- read.table("datatable.txt", nrows = 100)

classes <- sapply(initial, class)

tabAll <- read.table("datatable.txt", colClasses = classes)

Set nrows. This doesn’t make R run faster but it helps with memory usage. A mild overestimate

is okay. You can use the Unix tool wc to calculate the number of lines in a file.

**Textual Formats (special type of storing data)**

* **dumping** and **dputing** are useful because the resulting textual format is edit-able, and in the case of corruption, potentially recoverable.
* Unlike writing out a table or csv file, **dump** and **dput** preserve the metadata (sacrificing some
* readability), so that another user doesn’t have to specify it all over again.
* Textual formats can work much better with version control programs like subversion or git which can only track changes meaningfully in text files
* Textual formats can be longer-lived; if there is corruption somewhere in the file, it can be easier to fix the problem
* Textual formats adhere to the “Unix philosophy”
* Downside: The format is not very space-efficient

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

> x <- "foo"

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

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

> rm(x, y)

> source("data.R")

> y

a b

1 1 a

> x

[1] "foo"

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

> x <- "foo"

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

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

> rm(x, y)

> source("data.R")

> y

a b

1 1 a

> x

[1] "foo"

**Connections: Interfaces to the Outside World**

Data are read in using connection interfaces. Connections can be made to files (most common) 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

**File Connections**

> str(file)

function (description = "", open = "", blocking = TRUE, encoding = getOption("encoding"))

description is the name of the file

open is a code indicating

* “r” read only
* “w” writing (and initializing a new file)
* “a” appending
* “rb”, “wb”, “ab” reading, writing, or appending in binary mode (Windows)

In general, **connections** are powerful tools that let you navigate files or other external objects. In practice, we often don’t need to deal with the connection interface directly.

con <- file("foo.txt", "r")

data <- read.csv(con)

close(con)

is the same as

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

**Reading Lines of a Text File**

> con <- gzfile("words.gz")

> x <- readLines(con, 10)

> x

[1] "1080" "10-point" "10th" "11-point"

[5] "12-point" "16-point" "18-point" "1st"

[9] "2" "20-point"

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

**readLines** can be useful for reading in lines of webpages

## This might take time

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

x <- readLines(con)

> head(x)

[1] "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 Transitional//EN\">"

[2] ""

[3] "<html>"

[4] "<head>"

[5] "\t<meta http-equiv=\"Content-Type\"content=\"text/html;charset=utf-8

***L17 to 22 –Subsetting***

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

* **[** always returns an object of the same class as the original; can be used to select more than one element (there is one exception)
* **[[** is used to extract elements of a list or a data frame; it can only be used to extract a single element and the class of the returned object will not necessarily be a list or data frame
* **$** is used to extract elements of a list or data frame by name; semantics are similar to that of **[[**.

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

> x[1]

[1] "a"

> x[2]

[1] "b"

> x[1:4]

[1] "a" "b" "c" "c"

Use of great and small sign (logic)

> x[x > "a"]

[1] "b" "c" "c" "d"

> u <- x > "a"

> u

[1] FALSE TRUE TRUE TRUE TRUE FALSE

> x[u]

[1] "b" "c" "c" "d"

**Subsetting Lists**

> x <- list(foo = 1:4, bar = 0.6)

> x[1]

$foo

[1] 1 2 3 4

> x[[1]]

[1] 1 2 3 4

> x$bar

[1] 0.6

> x[["bar"]]

[1] 0.6

> x["bar"]

$bar

[1] 0.6

Subsetting multi-elements from a list

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

> x[c(1, 3)]

$foo

[1] 1 2 3 4

$baz

[1] "hello"

The [[ operator can be used with computed indices; $ 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

[1] 1 2 3 4 ## element ‘foo’ does exist

**Subsetting Nested Elements of a List**

The [[ can take an integer sequence.

> x <- list(a = list(10, 12, 14), b = c(3.14, 2.81))

> x[[c(1, 3)]]

[1] 14

> x[[1]][[3]]

[1] 14

> x[[c(2, 1)]]

[1] 3.14

**Subsetting a Matrix**

Matrices can be subsetted in the usual way with (i,j) type indices. Indices can also be missing.

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

> x[1, 2]

[1] 3

> x[2, 1]

[1] 2

> x[1, ]

[1] 1 3 5

> x[, 2]

[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 1 × 1 matrix. This behavior can be turned off by setting drop = FALSE.

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

> x[1, 2]

[1] 3

> x[1, 2, drop = FALSE]

[,1]

[1,] 3

Similarly, subsetting a single column or a single row will give you a vector, not a matrix (by default).

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

> x[1, ]

[1] 1 3 5

> x[1, , drop = FALSE]

[,1] [,2] [,3]

[1,] 1 3 5

**Partial Matching**

Partial matching of names is allowed with [[ and $.

> x <- list(aardvark = 1:5)

> x$a

[1] 1 2 3 4 5

> x[["a"]]

NULL

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

[1] 1 2 3 4 5

**Removing NA Values**

A common task is to remove missing values (NAs).

> x <- c(1, 2, NA, 4, NA, 5)

> bad <- is.na(x)

> x[!bad]

[1] 1 2 4 5

What if there are multiple things and you want to take the subset with no missing values?

> x <- c(1, 2, NA, 4, NA, 5)

> y <- c("a", "b", NA, "d", NA, "f")

> good <- complete.cases(x, y)

> good

[1] TRUE TRUE FALSE TRUE FALSE TRUE

> x[good]

[1] 1 2 4 5

> y[good]

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

> airquality[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

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

**Vectorized Operations**

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

> x <- 1:4; y <- 6:9

> x + y

[1] 7 9 11 13

> x > 2

[1] FALSE FALSE TRUE TRUE

> x >= 2

[1] FALSE TRUE TRUE TRUE

> y == 8

[1] FALSE FALSE TRUE FALSE

> x \* y

[1] 6 14 24 36

> x / y

[1] 0.1666667 0.2857143 0.3750000 0.4444444

> 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

Sample:

mean(subset(dset, Ozone > 31 & Temp > 90, select=Solar.R)$Solar.R)

Week 2 - John Hopkins Coursera

***L1 –L5 - Control structures***

Control structures in R allow you to control the flow of execution of the program, depending on runtime conditions. Common structures are:

* 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

Most control structures are not used in interactive sessions, but rather when writing functions or longer expresisons.

**if**

if(<condition>) {

## do something

} else {

## do something else

}

if(<condition1>) {

## do something

} else if(<condition2>) {

## do something different

} else {

## do something different

}

This is a valid if/else structure.

if(x > 3) {

y <- 10

} else {

y <- 0

}

So is this one.

**y <- if(x > 3)** {

10

} else {

0

}

Of course, the else clause is not necessary.

if(<condition1>) {

}

if(<condition2>) {

}

**For**

for loops take an interator variable and assign it successive values from a sequence or vector. For loops are most commonly used for iterating over the elements of an object (list, vector, etc.)

for(i in 1:10) {

print(i)

}

This loop takes the i variable and in each iteration of the loop gives it values 1, 2, 3, ..., 10, and then exits.

These three loops have the same behavior.

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

for(i in 1:4) {

print(x[i])

}

for(i in seq\_along(x)) {

print(x[i])

}

for(letter in x) {

print(letter)

}

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

**Nested for loops**

for loops can be nested.

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

for(i in seq\_len(nrow(x))) {

for(j in seq\_len(ncol(x))) {

print(x[i, j])

}

}

Be careful with nesting though. Nesting beyond 2–3 levels is often very difficult to read/understand

**While**

While loops begin by testing a condition. If it is true, then they execute the loop body. Once the loop body is executed, the condition is tested again, and so forth.

count <- 0

while(count < 10) {

print(count)

count <- count + 1

}

While loops can potentially result in infinite loops if not written properly. Use with care!

Sometimes there will be more than one condition in the test.

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

}

}

Conditions are always evaluated from left to right.

**Repeat**

Repeat initiates an infinite loop; these are not commonly used in statistical applications but they do have their uses. The only way to exit a repeat loop is to call break.

x0 <- 1

tol <- 1e-8

repeat {

x1 <- computeEstimate()

if(abs(x1 - x0) < tol) {

break

} else {

x0 <- x1

}

}

The loop in the previous slide is a bit dangerous because there’s no guarantee it will stop. Better to set a hard limit on the number of iterations (e.g. using a for loop) and then report whether convergence was achieved or not.

**next, return**

next is used to skip an iteration of a loop

for(i in 1:100) {

if(i <= 20) {

## Skip the first 20 iterations

next

}

## Do something here

}

**return** signals that a function should exit and return a given value

***L5 – L8 - Functions***

**Functions**

Functions are created using the function() directive and are stored as R objects just like anything else. In particular, they are R objects of class “function”.

f <- function(<arguments>) {

## Do something interesting

}

Functions in R are “first class objects”, which means that they can be treated much like any other R object. Importantly,

* Functions can be passed as arguments to other functions
* Functions 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.

**Function Arguments**

Functions have named arguments which 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
* Function arguments can be missing or might have default values

**Argument Matching**

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

> mydata <- rnorm(100)

> sd(mydata)

> sd(x = mydata)

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

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

> sd(na.rm = FALSE, mydata)

Even though it’s legal, I don’t recommend messing around with the order of the arguments too much, since it can lead to some confusion.

You can mix positional matching with matching by name. When an argument is matched by name, it is “taken out” of the argument list and the remaining unnamed arguments are matched in the order that they are listed in the function definition.

> **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, ...)

The following two calls are equivalent.

lm(data = mydata, y ~ x, model = FALSE, 1:100)

lm(y ~ x, mydata, 1:100, model = FALSE)

* Most of the time, named arguments are useful on the command line when you have a long argument list and you want to use the defaults for everything except for an argument near the end of the list
* Named arguments also help if you can remember the name of the argument and not its position on the argument list (plotting is a good example).

Function arguments can also be partially matched, which is useful for interactive work. The order of operations when given an argument is:

1. Check for exact match for a named argument

2. Check for a partial match

3. Check for a positional match

**Defining a Function**

f <- function(a, b = 1, c = 2, d = NULL) {

}

In addition to not specifying a default value, you can also set an argument value to NULL.

**Lazy Evaluation**

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

f <- function(a, b) {

a^2

}

f(2)

## [1] 4

This function never actually uses the argument b, so calling f(2) will not produce an error because the 2 gets positionally matched to a.

f <- function(a, b) {

print(a)

print(b)

}

f(45)

## [1] 45

## Error: argument "b" is missing, with no default

Notice that “45” got printed first before the error was triggered. This is because b did not have to be evaluated until after print(a). Once the function tried to evaluate print(b) it had to throw an error.

**The “...” Argument**

The ... argument indicate a variable number of arguments that are usually passed on to other

functions.

* ... is often used when extending another function and you don’t want to copy the entire argument list of the original function

myplot <- function(x, y, type = "l", ...) {

plot(x, y, type = type, ...)

}

* Generic functions use ... so that extra arguments can be passed to methods (more on this later).

> mean

function (x, ...)

UseMethod("mean")

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

> args(paste)

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

> args(cat)

function (..., file = "", sep = " ", fill = FALSE,

labels = NULL, append = FALSE)

**Arguments Coming After the “...” Argument**

One catch with ... is that any arguments that appear after ... on the argument list must be named explicitly and cannot be partially matched.

> args(paste)

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

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

[1] "a:b"

> paste("a", "b", se = ":")

[1] "a b :"

***L9 – L 11 Scoping Rules***

**A Diversion on Binding Values to Symbol**

How does R know which value to assign to which symbol? When I type

> lm <- function(x) { x \* x }

> lm

function(x) { x \* x }

how does R know what value to assign to the symbol lm? Why doesn’t it give it the value of lm that is in the stats package?

When R tries to bind a value to a symbol, it searches through a series of environments to find the appropriate value. When you are working on the command line and need to retrieve the value of an R object, the order is roughly

1.Search the global environment for a symbol name matching the one requested.

2.Search the namespaces of each of the packages on the search list

The search list can be found by using the search function.

> search()

[1] ".GlobalEnv" "package:stats" "package:graphics"

[4] "package:grDevices" "package:utils" "package:datasets"

[7] "package:methods" "Autoloads" "package:base"

**Binding Values to Symbol**

·The global environment or the user’s workspace is always the first element of the search list and the base package is always the last.

·The order of the packages on the search list matters!

·User’s can configure which packages get loaded on startup so you cannot assume that there will be a set list of packages available.

·When a user loads a package with library the namespace of that package gets put in position 2 of the search list (by default) and everything else gets shifted down the list.

·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.

**Scoping Rules**

The scoping rules for R are the main feature that make it different from the original S language.

·The scoping rules determine how a value is associated with a free variable in a function

·R uses lexical scoping or static scoping. A common alternative is dynamic scoping.

·Related to the scoping rules is how R uses the search list to bind a value to a symbol

·Lexical scoping turns out to be particularly useful for simplifying statistical computations

**Lexical Scoping or Static Scoping**

Consider the following function.

f <- function(x, y) {

x^2 + y / z

}

This function has 2 formal arguments x and y. In the body of the function there is another symbol z. In this case z is called a free variable. The scoping rules of a language determine how values are assigned to free variables. Free variables are not formal arguments and are not local variables (assigned insided the function body).

Lexical scoping in R means *that 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; it is possible for an environment to have multiple “children”

·the only environment without a parent is the empty environment

·A function + an environment = a closure or function closure.

Searching for the value for a free variable:

·If the value of a symbol is not found in the environment in which a function was defined, then the search is continued in the parent environment.

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

·After the top-level environment, the search continues down the search list until we hit the empty environment. If a value for a given symbol cannot be found once the empty environment is arrived at, then an error is thrown.

Why does all this matter?

·Typically, a function is defined in the global environment, so that the values of free variables are just found in the user’s workspace

·This behavior is logical for most people and is usually the “right thing” to do

·However, in R you can have functions defined inside other functions

-Languages like C don’t let you do this

·Now things get interesting — In this case the environment in which a function is defined is the body of another function!

make.power <- function(n) {

pow <- function(x) {

x^n

}

pow

}

This function returns another function as its value.

> cube <- make.power(3)

> square <- make.power(2)

> cube(3)

[1] 27

> square(3)

[1] 9

Exploring a Function Closure - What’s in a function’s environment?

> ls(environment(cube))

[1] "n" "pow"

> get("n", environment(cube))

[1] 3

> ls(environment(square))

[1] "n" "pow"

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

[1] 2

**Lexical vs. Dynamic Scoping**

y <- 10

f <- function(x) {

y <- 2

y^2 + g(x)

}

g <- function(x) {

x\*y

}

What is the value of

f(3)

·With lexical scoping the value of y in the function g is looked up in the environment in which the function was defined, in this case the global environment, so the value of y is 10.

·With dynamic scoping, the value of y is looked up in the environment from which the function was called (sometimes referred to as the calling environment). -In R the calling environment is known as the parent frame

·So the value of y would be 2.

When a function is defined in the global environment and is subsequently called from the global environment, then the defining environment and the calling environment are the same. This can sometimes give the appearance of dynamic scoping.

> g <- function(x) {

+ a <- 3

+ x+a+y

+ }

> g(2)

Error in g(2) : object "y" not found

> y <- 3

> g(2)

[1] 8

**Other languages that support lexical scoping**

·Scheme

·Perl

·Python

·Common Lisp (all languages converge to Lisp)

**Consequences of Lexical Scoping**

·In R, all objects must be stored in memory

·All functions must carry a pointer to their respective defining environments, 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.

**Application: Optimization**

Why is any of this information useful?

·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)

·However, 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

**Maximizing a Normal Likelihood**

Write a “constructor” function

make.NegLogLik <- function(data, fixed=c(FALSE,FALSE)) {

params <- fixed

function(p) {

params[!fixed] <- p

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)

}

}

Note: Optimization functions in R minimize functions, so you need to use the negative log-likelihood.

> set.seed(1); normals <- rnorm(100, 1, 2)

> nLL <- make.NegLogLik(normals)

> nLL

function(p) {

params[!fixed] <- p

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)

}

<environment: 0x165b1a4>

> ls(environment(nLL))

[1] "data" "fixed" "params"

**Estimating Parameters**

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

mu sigma

1.218239 1.787343

Fixing σ = 2

> nLL <- make.NegLogLik(normals, c(FALSE, 2))

> optimize(nLL, c(-1, 3))$minimum

[1] 1.217775

Fixing μ = 1

> nLL <- make.NegLogLik(normals, c(1, FALSE))

> optimize(nLL, c(1e-6, 10))$minimum

[1] 1.800596

**Plotting the Likelihood**

nLL <- make.NegLogLik(normals, c(1, FALSE))

x <- seq(1.7, 1.9, len = 100)

y <- sapply(x, nLL)

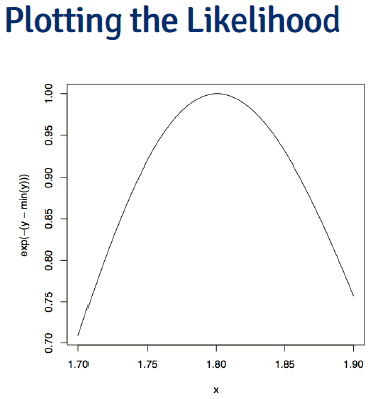
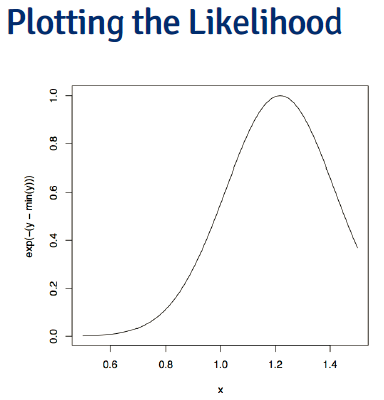
plot(x, exp(-(y - min(y))), type = "l")

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")



**Lexical Scoping Summary**

Objective functions 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 cleand up

Reference: Robert Gentleman and Ross Ihaka (2000). “Lexical Scope and Statistical Computing,” JCGS, 9, 491–508.

***L12 – Coding Standards***

**Coding Standards for R**

1.Always use text files / text editor

2.Indent your code

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

4.Limit the length of individual functions

***L13 – Dates and Times***

**Dates and Times in R**

R has developed a special representation of dates and times

·Dates are represented by the Date class

·Times are represented by the POSIXct or the POSIXlt class

·Dates are stored internally as the number of days since 1970-01-01

·Tmes are stored internally as the number of seconds since 1970-01-01

**Dates in R**

Dates are represented by the Date class and can be coerced from a character string using the as.Date() function.

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

>x

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

>unclass(x)

## [1] 0

>unclass(as.Date("1970-01-02"))

## [1] 1

**Times in R**

Times are represented using the POSIXct or the POSIXlt class

·POSIXct is just a very large integer under the hood; it use a useful class when you want to store times in something like a data frame

·POSIXlt is a list underneath and it stores a bunch of other useful information like the day of the week, day of the year, month, day of the month

There are a number of generic functions that work on dates and times

·weekdays: give the day of the week

·months: give the month name

·quarters: give the quarter number (“Q1”, “Q2”, “Q3”, or “Q4”)

Times can be coerced from a character string using the as.POSIXlt or as.POSIXct function.

>x <- Sys.time()

>x

## [1] "2013-01-24 22:04:14 EST"

>p <- as.POSIXlt(x)

>names(unclass(p))

## [1] "sec" "min" "hour" "mday" "mon"

## [6] "year" "wday" "yday" "isdst"

>p$sec

## [1] 14.34

You can also use the POSIXct format.

>x <- Sys.time()

>x ## Already in ‘POSIXct’ format

## [1] "2013-01-24 22:04:14 EST"

>unclass(x)

## [1] 1359083054

>x$sec

## Error: $ operator is invalid for atomic vectors

>p <- as.POSIXlt(x)

>p$sec

## [1] 14.37

Finally, there is the **strptime** function in case your dates are written in a different format

datestring <- c("January 10, 2012 10:40", "December 9, 2011

>x <- strptime(datestring, "%B %d, %Y %H:%M")

>x

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

>class(x)

## [1] "POSIXlt" "POSIXt"

I can never remember the formatting strings. Check **?strptime** for details.

**Operations on Dates and Times**

You can use mathematical operations on dates and times. Well, really just + and -. You can do comparisons too (i.e. ==, <=)

>x <- as.Date("2012-01-01")

>y <- strptime("9 Jan 2011 11:34:21", "%d %b %Y %H:%M:%S")

>x-y

## Warning: Incompatible methods ("-.Date",

## "-.POSIXt") for "-"

## Error: non-numeric argument to binary operator

>x <- as.POSIXlt(x)

>x-y

## Time difference of 356.3 days

Even keeps 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

**Assigment**

pollutantmean <- function(directory, pollutant, id = 1:332) {

## 'directory' is a character vector of length 1 indicating

## the location of the CSV files

## 'pollutant' is a character vector of length 1 indicating

## the name of the pollutant for which we will calculate the

## mean; either "sulfate" or "nitrate".

## 'id' is an integer vector indicating the monitor ID numbers

## to be used

## Return the mean of the pollutant across all monitors list

## in the 'id' vector (ignoring NA values)

## NOTE: Do not round the result!

data = numeric()

for (i in id) {

newRead = read.csv(paste(directory, "/", formatC(i, width = 3, flag = "0"),

".csv", sep = ""))

data = c(data, newRead[[pollutant]])

}

return(mean(data, na.rm = TRUE))

}

Week 3 - John Hopkins Coursera

Week 4 - John Hopkins Coursera

Swirl Courses

<http://swirlstats.com/>

This is the only step that you will repeat every time you want to run swirl. First, you will load the package using the library() function. Then you will call the function that starts the magic! Type the following, pressing Enter after each line:

> library("swirl")

> swirl()

<https://github.com/swirldev/swirl_courses#swirl-courses>

| You can exit swirl and return to the R prompt (>) at any time by pressing the Esc key. If

| you are already at the prompt, type bye() to exit and save your progress. When you exit

| properly, you'll see a short message letting you know you've done so

| When you are at the R prompt (>):

| -- Typing skip() allows you to skip the current question.

| -- Typing play() lets you experiment with R on your own; swirl will ignore what you do...

| -- UNTIL you type nxt() which will regain swirl's attention.

| -- Typing bye() causes swirl to exit. Your progress will be saved.

| -- Typing main() returns you to swirl's main menu.

| -- Typing info() displays these options again.

Lessons

1: Basic Building Blocks

2: Workspace and Files

3: Sequences of Numbers

4: Vectors

5: Missing Values

6: Subsetting Vectors

7: Matrices and Data Frames

8: Logic

9: Functions

10: lapply and sapply

11: vapply and tapply

12: Looking at Data

13: Simulation

14: Dates and Times

15: Base Graphics