R-Programming

# Objective of R-Programming

* Have solid grasp of R as programming language.
* Get you into R programming

# Overview and History of R

## History of S

* R is a dialect of S
* S was developed by John Chambers at Bell Labes in 1976 for internal statistical analysis
* Version 3 was written in 1988 and released with book Statistical Models in S (white book)
* Version 4 of S language was released in 1998 and is version we use today. The book Programming with Data (the green book) documents this version
* Insightful sells S currently under S-PLUS; insightful was acquired by TIBCO

Key principal: S is interactive environment where users did not think of themselves as programming. As needs became clearer and sophistications increased, they could slide into programming. They would transition from user to programmer. (John Chambers)

## History of R

* Created in 1991 in New Zealand
* Licensed under GNU General Public License making it free
* Public mailing lists - R-Help and R-devel
* R Core Group formed in 1997 - controls source code
* 2013: R version 3.0.2 released

## Features of R

* Syntax very similar to S
* Runs on any standard OS
* Frequent releases (annual + bugfix); active development
* Functionality divided into modular packages
* Graphics capabilities very sophisticated; useful for interactive work and developing new tools
* Very active and vibrant user community; R-help, R-devel, stack overflow
* It’s free!

## Drawbacks of R

* Based on 40 year old technology
* Little built in support for dynamics or 3-D graphics
* Based on consumer demand, if no one feels like implementing favourite method it’s your job
* Objects must be stored in physical memory (there are advancements to deal with)

## Design of R System

* 2 Conceptual parts: base R system AND everything else
* Base R contains **base** package required to run R and contains the most fundamental functions
* There are about 4k packages on CRAN
* there are many on Bioconductor project (<http://bioconductor.org>)

## R Resource

* From cran.r-project.org
* Introduction to R
* Writing R Extensions (packages)
* R Data Import/Export
* R Installation and Administration (for building R from sources)
* R Internals (not for faint of heart!!)

# Getting Help

## Finding Answers

* Search archive of forum
* Search the Web
* Read the manual
* Read a FAQ
* Try inspection or experimentation
* Ask a skilled friend
* If a programming - try reading the source code

## Asking Questions

* Need to understand via email people don’t have background, are busy and their time is limited
* Instructor may be able to help but may not be able to answer all questions (use other resources)
* Let other people know that you’ve done the previous things already (done your homework)
* Things to provide:
  + What steps will reproduce problem
  + What is expected output
  + What do you see instead
  + What version of the product (R, packages) are you using (if you are using latest version, mention that)
  + What OS

## Subject Header

* Stupid: “Help! Can’t fit linear model!”
* Smart: “R 3.0.2 lm() function produces seg fault with large data frame, Mac OS X 10.9.1”
* Smarter: “R 3.0.2 lm() function on Mac OS X 10.9.1 -- seg fault on large data frame”

## Things to Do When Asking a Forum

* Describe goal, not steps
* Be explicit about question
* Provide minimum amount of info necessary (volume is not precission)
* Be courteous
* Follow up with solution (if found)

## Things NOT to do

* Claim you’ve found a bug
* Grovel as sub for doing homework
* Post homework question
* Email multiple mailing lists at once
* Ask other to debug broken code without giving hint of the problem to look for

## Places to Turn

* Class discussion board; your fellow students
* [r-help@r-project.org](mailto:r-help@r-project.org)
* Other project-specific mailing lists (see “How to ask questions the smart way” -Eric Raymond)

# R Objects and Attributes

## R has five basic or “atomic” classes of objects

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

## Most basic object is a vector

* A vector can only contain objects of the same class
* One exception is a *list*, which is represented as vector but can contain objects of different classes
* Empty vectors can be created with **vector()** function

## Numbers

* Treated as double precision numbers
* To explicitly want a number use **L** suffix
* Special number **Inf** representing infinity. e.g. 1 / Inf = 0
* **NaN** is an undefined value, e.g. 0/0 = NaN

## Attributes

* names, dimnames
* dimensions (e.g. matrices, arrays)
* class (e.g. numeric)
* length
* other user-defined attributes/metadata
* Can be accessed using **attributes()** function

## Basics - Files, Directory and Console

|  |  |
| --- | --- |
| Command | Description |
| getwd() | prints working directory |
| dir() or list.files() | shows content of directory |
| source("mycode.R") | loads r file |
| {Ctrl+L} | Clear console |
| {Ctrl+I) | Apply preferences in RStudio |
| setwd("~/DataScience/2-RProgramming/specdata") | changes working directory |
| search() | Shows all packages loaded into R environment |
| dir.create(“directory”) | create directory as child folder in working directory |
| dir.create(file.path("testdir2", "testdir3"),recursive = TRUE) | creates recursive directory |
| unlink(“testdir2”, recursive = TRUE) | removes recursive directory |
| file.create(“file”) | creates file in working directory |
| file.exists(“file”) | logical test to see if file exists |
| file.info(“file”) | return metadata about file |
| file.rename(“old”,”new”) | renames file |
| file.remove(“file”) | removes file |
| file.path(“file”) | returns relative path of file |
| file.path(“folder1”,”folder2”) | constructs directory path independent of OS |
| options(digits =6) | sets the default digits to round to |

## Obtaining R Packages

|  |  |
| --- | --- |
| Command | Description |
| a <- available.packages() | gets list of available packages |
| head(rownames(a),3) | shows first three |
| install.packages(“devtools”) | install package |
| install.packages(c(“devtools”, “solidify”) | install multiple packages |
| library(devtools) | load library |
|  |  |
|  |  |

## Sequences of Numbers

|  |  |
| --- | --- |
| Command | Description |
| ls() | shows objects in local workspace |
| object.size(object) | shows how much memory an object is taking up in bytes |
| **Vectors**  x <- vector()  x <- c(0.5, 0.6)  x <- 9:29 | creates empty vector, logical by default  creates numeric vector  creates integer vector |
| identical(vect,vect2) | logical test if two vectors are the same |
| **Using Seq**  x <- seq(0, 10, by=0.5)  x <- seq(0, 30, length=30)  y <- seq\_along(x) | creates numeric vector from 0 to 10, incremented by 0.5  creates numeric vector from 0 to 10, with 30 elements (incremented evenly)  creates vector of same length of x, incremented by 1 by default |
| **Using Rep**  x <- rep(c(0,1,2), times=10)  x <- rep(c(0,1,2), each=10) | creates vector composed of sequence 0,1,2 repeated 10 times  creates vector with 0 repeated 10 times, then 1 repeated 10 times, then 2 repeated 10 times |
| **Mixing Objects**  y <- c(1.7,”a”)  y <- c(TRUE, 2)  y <- c(“a”, TRUE) | *coercion occurs so every object is of same class*  character, 1.7 coerced to “1.7”  numeric, TRUE coerced to 1  character, TRUE coerced to “TRUE” |
| **Explicit coercion**  x <- 0:6  as.character(x) | declares integer vector  converts all components in x to character  *will return NaN if can’t convert (e.g. “a” to numeric)* |
| **Lists**  x <- list(1, “a”, TRUE, 1 + 4i) | declares lists with elements of different classes |
|  |  |
| attributes(x) | return attributes of object x |
| class(x) | returns class of object |
| is.na(x) | returns TRUE if missing |
| is.nan(x) | returns TRUE if undefined mathematical component |
| names(x) <- c(“foo”, “bar”, “norf”) | gives name to each element of vector |
| x <- list (a =1 , b =2, c =3) | declares list with named elements |
| length(x) | returns number of elements in object |
| (my\_char, collapse = “ “) | Pastes elements of vector together, using “ “ as delimiter of elements |
| paste(vector1,vector2,sep=” “) | Pastes each element of first vector with element of second element, delimited by sep (result will be vector with same length of vector1) |
| unique(vector) | returns elements of vector with duplicates removed |
| intersect(v1,v2) | returns objects found in both vectors |
| sort(avgstepsper5m,decreasing=TRUE) | Sort a vector in descending order |
| attributes(avgstepsper5m[1])$names | Returns attribute of one element of an array vector |

## Matrices

|  |  |
| --- | --- |
| Command | Description |
| m <- matrix (nrow = 2, ncol = 3) | declares empty matrix with 2 rows and 3 columns  *matrices are constructed column-wise* |
| m <- matrix(rnorm(4),nrow = 2, ncol = 2) | creates matrix composed of four random numbers within 2 rows and 2 columns |
| dim(m) | returns dimensions of matrix, rows followed by columns |
| m <- 1:10  dim(m) <- c(2,5) | assign dimension to vector to transform to matrix |
| x <- 1:3  y <- 10:12  cbind(x,y)  rbind(x,y) | create matrix by binding x vector to first column and y vector to second column  create matrix by binding x vector to first row and y vector to second row |
| dimnames(m) < list(c(“a”,b” |  |

# Factors

Factors used to represent categorical data. Can be ordered or unordered. Factors are like integer vector where each integer has *label*.

* Factors are treated specially be modelling functions like **lm()** and **glm()**
* Better than using integers because factors are self-describing; having a variable with values “Male” and “Female” is better than using variable with values 1 & 2
* The order of levels can be using **levels** argument. This can be important to linear modelling because first level used as baseline level.

## Attributes

* names, dimnames
* dimensions (e.g. matrices, arrays)
* class (e.g. numeric)

|  |  |
| --- | --- |
| Command | Description |
| x < factor(c(“yes”,”yes”,”no”,”yes”,”no”)) | Declare factor with two levels |
| table(x) | returns labels in table counted and grouped by label |
| unclass(x) | shows how factors are treated in R as integers |
| x <- factor(c("yes","yes","no"), levels = c("yes", "no")) | explicitly set order of levels so “yes” is first |
| interactions(f1,f2) | Shows how factors can be combined |

# Missing Values

* Missing values denoted by NA
* Undefined mathematical operations denoted by NaN

|  |  |
| --- | --- |
| Command | Description |
| is.na(my\_data) | Logical test of values in my\_data if they are NA or not |

# Data Frames

* Used to store tabular data
* Special type of list where every element of list has same length
* Each element of list can be thought of as column, and length of each element of number of rows
* Data frames can store different classes, unlike matrices
* Data frames have special attribute called **row.names**
* Data frames usually created by calling **read.table()** or **read.csv()**
* Can be converted to matrix by calling data.matrix()

|  |  |
| --- | --- |
| Command | Description |
| x <- data.frame(foo = 1:4, bar = c(T,T,F,F)) | Declare data frame (table) with first column “foo” and second column “bar” and populated with values |
| nrow(x) | returns number of rows in data frame |
| ncol(x) | returns number of columns in data frame |
| str() | returns metadata of data.frame |
| my\_data <- data.frames(patients, my\_matrix) | creates data frame where the first columns come from patients and subsequent columns come from my\_matrix |
| colnames(my\_data) <- cnames | assigns column names to my\_data from cnames (must be same length) |
| df <- rbind(df, read.csv(files)) | add rows from csv file to existing data frame |
| RawCSV <- read.csv(files\_list[1], colClasses = "character") | Creates data.frame from csv file |
| myVar <- df[[colvar]]  myVar <- df$colname  myVar <- df[,columnnum] | creates vector from data frame where colvar is a column name stored in a variable  creates vector from data frame with explicit column name  creates vector from data frame with column number |
| outcome[,11] <- as.numeric(outcome[,11]) | coerce one column two numeric |
| data[order(mpg, -cyl),] | orders data frame by mpg (ascending), cyl (descending) variables |
| names(df) <- make.names(cnames[[1]]) | set names of data frame from vector in list, using make.names function to coerce values to valid header names |
| counties <- data.frame(fips = c("24510","06037"),county = c("Baltimore City","Los Angeles County")) | create data.frame with rows |

# Reading Data

* **read.table** function is one of most commonly used functions for reading data
  + **file** name of file or connection
  + **header**, logical indicator if file has header
  + **sep**, string indicating how columns are separated (default is “”, white space, one or more spaces, tabs, newlines or carriage returns)
  + **colClasses**, a character vector indicating class of each column in dataset
  + **nrows**, the number of rows in dataset
  + **comment.char**, a character string indicating comment character (default # symbol, this is ignored)
  + **skip**, number of lines to skip from beginning
  + **stringAsFactors**, should character variables be coded as factors? defaults to true
* For smallish datasets you can usually use just file, e.g. data <- **read.table**(“foo.text”), R will automatically**:**
  + skip lines that begin with #
  + figure out how many rows there are (to allocate memory)
  + Figure what type of variable in each column of table, Telling R all these things directly makes R run fastor and more efficiency
  + read.csv is identify to read.table except that default separator is a comma

|  |  |
| --- | --- |
| Command | Description |
| con <- file(“foo.text”, “r”) | Establishes read connection to text file |
| data <- read.csv(con) | Reads entire file |
| x <- readLines(con, 10) | Reads first 10 lines of file only |
| close(con) | closes connection |
| writeLines | Takes a character vector and writes each element one line at a time to a text file |
| con <- url([http://www.jhsph.edu,”r](http://www.jhsph.edu,)”) | Establishes connection to website |
| x <- readLines(con) |  |
| head(x) | Gets header info from website |
| print(delay,n=10000) | Printing data from an object with 10000 lines. Useful when wrapped in tbl\_df with default of only 10 lines that are printed. |

# Reading Large Tables

With much larger datasets, the following will make life easier

* read help pages for read.table, which contains many hints
* Make a rough calculation of memory required to store dataset. If larger than amount of RAM on computer, stop
* Set comment.char = “ “ if there are no commented lines in your file
* Use colClasses argument to specify class, a fast way to do this is:
  + initial <- read.table(“datatable.txt”, nrows =100)
  + classes <- sapply(initial, class)
  + tabAll = read.table(“datatable.txt”,colClasses = classes)
* set nrows. Doesn’t make R run faster but helps with memory usage. Mild overestimate is okay. Use Unix **wc** to calculate number of lines in a file.
* Know thy system:
  + How much memory
  + What other applications are in use
  + Are other users logged into system
  + What OS
  + Is it 32 bit or 64 bit? 64 bit can access more memory
* Calculating memory requirements: e.g. data frame with 1.5 million rows and 120 columns, all of which are numeric. How much memory is required?
  + 1500000 x 120 x 8 bytes/numeric
  + =1440000000 bytes
  + =1440000000 /2^20 bytes/MB
  + 1,3272.29 MB
  + 1.34 GB
  + Need twice as much to read as much in so 2.68 GB

# Textual Formats

* **dumping** and **dputing** are useful because of resulting textual format is edit-able, and in the case of corruption, potentially recoverable (otherwise it will be stored in binary)
* unlike writing out of a table or csv file, **dump** and **dput** preserve the metadata (sacrificing some readability), so that another user doesn’t have to specify again
* Textualformats can work much better with version control programs like git which can only trac changes meaningfully in text files
* Textual formats can be longer-lives; if there is corruption somewhere in the file it can be easeier to fix the problem,
* Textual format adhere to “Unix philosophy”
* Downside: The format is not very space-efficient
* dput-ting R Objects
  + another way to pass data around is by deparsing R object with **dput** and reading it back in using **dget**
* dumping R Objects
  + multiple objects can be deparsed using the dump function and read back in using *source*

|  |  |
| --- | --- |
| Command | Description |
|  |  |
| y <- data.frame(a=1,b=”a”)  dput(y)  dput(y,file=”y.R”)  new.y <- dget(“y.R”) | constructs table in standard output  constructs table to file y.R  new.y table constructed from file |
| x <- “foo”  y <- data.frame(a=1,b=”a”)  dump(c(“x”,”Y”, file = “data.R”) | declares variable x  declares table y  dumps both objects x and y to file data.R |
| source(“data.R”) | gets both objects from dump file |
|  |  |

# Interfaces to the Outside World

* Data are read in using connection interfaces, including:
  + **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

# Subsetting

* There are a number of operations that can be used to extract subsets of R objects
* [ always return an object of same class as original; can be used to select more than one element (one exception)
* [[ is used to extract elements of a list or data frame; can only be used to extract single element and class of returned object will not necessarily be same as list or data frame
* $ used to extract elements of a list or data frame by name; semantics are similar to that of [[

|  |  |
| --- | --- |
| Command | Description |
| **subsetting vectors**  x <- c(“a”,”b”,”c”,”c”,”d”,”a”)  x[1]  x[1:4]  x[x > “a”]  u <- x > “a”  x[u]  x[c(2,10)]  x[c(-2,-10)] | extract first element of x  extracts first four elements of x  extracts elements of x after “a”  creates vector with logical ements based on each element of x being after “a”  extracts elements of x after “a”  extracts the 2nd and 10th elements of x  exctracts all elements of x except the 2 and 10th |
| **subsetting lists**  x <- list(foo = 1:4, bar = 0.6, baz = “hello”)  x[1]  x[[1]]  x$bar  x[c(1,3)]  name <- “foo”  x[[name]] ##computed index for ‘foo’  x <- list (a = list(10, 12, 14), b = c(3.14, 2.81))  x[[c(1,3)]] | extracts first element of x as list  extracts first element of x as vector  extracts element labelled “bar” from list  extracts multiple element of list  extracts element related with name  extracts third element of first list |
| **subsetting matrices**  x <- matrix(1:6, 2,3)  x[1, 2]  x[1,]  x[1, 2, drop = FALSE] | extracts element in first row and second column  extracts first row of matrix  extracts first row of matrix, preserving dimension (object remains matrix) |
| **subsetting with names - partial matching**  x <- list (aardvark = 1:5)  x$a  x[[“a”]]  x[[“a”, exact = FALSE] | extracts element with name starting with a  returns NULL (looks for exact match)  extracts element with name starting with a |
| **subsetting - removing missing values**  x <- c(1,2,NA,4, NA, 5)  bad <- is.na(x)  x[!bad]  y <- c(“a”, “b”, NA, “d”, NA, “f”)  good <- complete.cases(x, y) | bad is vector of logical elements on which in x are missing  returns vector which are not missing (NAs)  returns where both x and y are not missing |
| my\_data <- sample(c(y,z),100)) | takes random sample of y and z concatenated of 100 elements |
| corr\_sample[corr\_sample$nobs > 100, ] | returns subset of data frame where column meets condition |
| ints <- sample(10) | random sample of integers from 1 to 10 without replacement |
| **subsetting data frames**  subset(RawCSV, !is.na(RawCSV[columnvar]))  subset(RawCSV, !is.na(y)) | subsets data frame based on column condition where columnvar is stored in variable  subsets data frame based on column condition where y is explicitly named column |
| **Subsetting data frame**  data[, c(2,7, colnum[,1])]  subset(data[,c(2,7)], var = x) | returns 2nd, 7th and another column based on colnum variable  returns 2nd and 7th columns where var = x |
| merge(NEI,counties)  [,c("year","county","Emissions")] | subset using column names |
|  |  |

# Vectorized Operations

Things can happen in parallel, making code more efficient, concise and easier to read

|  |  |
| --- | --- |
| Command | Description |
| x <- 1:4; y <- 6:9  x + y  x > 2  x <- matrix (1:4, 2, 2)  y <- matrix (rep(10,4), 2, 2)  x \* y  x %\*% y | adds each element of vectors in same place  for each element in x, returns result of logical test  performs element-wise multiplication  performs true matrix multiplication |

# Control Structures in R

* if, else: testing a condition
* for: executing a loop a fixed number of times
* while: executing a loop *while* a condition is true
* repeat: execute an infinite loop
* break: break the execution of a loop
* next: skip an iteration of a loop
* return: exit a function

Most not used in interactive session, but rather when writing functions or longer expressions

For loops take an interator variable and assign it successive values from a sequence or vector.

|  |  |
| --- | --- |
| Command | Description |
| if(<condition>) { ## do something  } else if {  ## do something else  } else {  ## do something else  } | if/else control structure |
| TRUE & c(TRUE, FALSE, FALSE)  ## TRUE FALSE FALSE  TRUE && c(TRUE, FALSE, FALSE)  ## TRUE | Evaluates AND across vector  Evaluates AND for first member of vector |
| isTRUE( argument ) | If argument evaluates to TRUE, returns TRUE |
| xor(5 ==7, !FALSE) | exclusive OR function, exactly one must evaluate to TRUE to return TRUE |
| which(logical vector) | returns indices of the vector that are TRUE |
| any(logical vector) | returns TRUE if one or more element is TRUE |
| all(logical vector) | returns TRUE if every element is TRUE |
| for(I in 1:10) {  print(i)  }  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]) | basic for operator  seq\_along creates integer sequience of length of x |
| x <- matrix(1:6,2,3)  for(I in seq\_len(nrow(x))) {  for(j in seq\_len(ncol(x))) {  print(x[I,j])  }  } | nested for loop  be careful with nesting. Beyond 2-3 levels is often very difficult to read/understand. |
| count <- 0  while(count < 10) {  print(count)  count <- count + 1  } | while loop. will execute while condition is true  can potentially result in infinite loops if not written properly. use with care! |
| 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  + }  + } | testing more than one condition |
| x0 <- 1  tol <- 1e-8  repeat {  x1 <- computeEstimate()  if(abs(x1 - x0) < tol) {  break  } else {  x0 <- x1  }  } | **repeat** initiates infinite loop  **break** is only way to exit infinite loop  repeat is a bit dangerous. better to set hard limit on number of iterations (e.g. using a for loop) and then report whether convergence was achieved or not |
| for (I in 1:100) {  if(I <= 20) {  ## skip the first 20 iterations  next  }  ##do something  } | **next** is used to skip an iteration of a loop |

# Writing Functions

* Want to write functions in a text file
* Possible to write in command line, but not recommended
* Later, want to put into package
* Work with r script

# Functions

Functions represents most powerful aspect of R.

Created using function() directive and stored as R objects just like anything else (R objects of class “function”)

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

* Functions can be passed as argument to other functions
* Functions can be nested, so they can define a function inside another fuction. Return value of function is last in

## Function Arguments

Functions have *named arguments* which potentially have *default values*

* The format arguments are arguments included in function definition
* The formals function returns a list of all format arguments of a function
* Not every function call in R makes use of all formal arguments
* Function arguments can be missing or might have default values

## Argument Matching

R functions arguments can be matched positionally or by name. E.g.

* sd(mydata, FALSE) or
* sd(na.rm = FALSE, x = mydata)

Don’t mess around with order of arguments, since it can lead to confusion.

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

Following two are equivalent:

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

Named argumetns are useful on command line when you have long argument list and want to use defaults for everything except for an argument near end.

Named arguments also help if you can remember name of argument and not position (plotting is good example)

Can also be partially matched, useful for interactive work. Order of argument is:

1. Check for exact match for a named argument
2. Check for partial match
3. Check for positional match

## Lazy Evaluation

Arguments to function 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.

## The “…” Argument

The … argument indicates 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 entire argument list to the original function

myplot <- function (x , y, type = “1”, …) {  
 plot(x, y, type = type, …)  
}

Generic functions use … so that extra arguments can be passed to methods

Also necessary when arguments passed to function cannot be known in advance.

Example: >args(cat)

function(… , file = “”, sep = “ “, fill = FALSE, labels = NULL, append = FALSE)

Any arguments that appear after … on arguments MUST be named explicitly and cannot be partially matched.

|  |  |
| --- | --- |
| Command | Description |
| ?function or help(function) | opens help file for function |
| function | typing function without parentheses prints source code of function |
| args(function) | Displays arguments of function |
| ls(environment(function)) | lists what’s in a function environment |
| get(“n”, environment(function) | returns value of n in given function |
| "%binary\_operator%" <- function(left, right){ left \* right + 1 } | create binary operator which operators like left %binary\_operator% right |
| invisible(x) | allows you to assign output of function to variable without printing |

# Scoping Rules

## A Diversion on Binding Values to Symbol

How does R know which value to assign to which symbol?

e.g. lm <- function (x) {x \* x}

How does R know what value to assign to lm? As function lm already defined.

When R tried to do so, it searches through series of environments to find appropriate value. When working on command line, and need value of an R object, order is roughly:

1. Search global environment for a symbol matching requested - your workspace
2. Search namespaces of each packages on search list

## Binding Values to Symbol

* global environment of user’s workspace is always first element of search list and *base* package is always last
* order of packages on search list matters!
* users can configure which packages get loaded on startup, so cannot assume there will be a set list of packages available
* when user loads package with *library* namespace package gets put in position 2 by default and rest gets shifted down
* 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

Scoping rules for R main feature different from original S language

* scoping rules determine how value is associated with free variable in function
* R uses lexical scoping or static scoping. A common alternative is dynamic scoping
* Related to scoping rules is how R uses search *list*  to bind a value to a symbol
* Lexical scoping turns out to be particularly useful for simplifying statistical computations

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

It has 2 formal arguments x and y. In function here is another symbol z. In this case z is called a *free variable* Scoping rules of language determine how values are assigned to free variables. They are not formal arguments and not local variables (assigned inside function body).

Lexical scoping in R means: *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 2.14 might be its value
* Every environment has 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* - key to a lot of different operations

Searching for value for a free variable:

* If value of symbol not found in environment in which function was defined, then search is continued in parent environment
* The search continues down sequence of parent environments until we hit top-level environment; this usually the global environment (workspace) or namespace of a package
* After top-level environment search continued down search list until we hit empty environment, if a value for a given symbol cannot be found once the empty environment is arrived at, then error is thrown

Why does all this matter?

* Typically, function is defined in global environment, so values of free variables are just found in user’s workspace
* This behaviour is logical for most people and is usually “right thing”
* However, in R you can have functions defined inside other fucntions
  + Languages like C don’t let you do this
* Now things get interesting in this case the environment in whicha function is defined is the body of another function! This is a constructor function.

## Lexical vs Dynamic Scoping

Given:  
y <- 10  
f <- function(x) {  
 y <- 2  
 y^2 + g(x)  
}  
g <- function(x) {  
 x\*y  
}

* 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, value of y is looked up in the environment from which the function was *called* (somethings referred to as the *calling environment)*
  + I R the calling environment is known as *parent frame*
* So value of y would be 2.
* When function is *defined* in global environment and then *called* from global environment then the defining environment and calling environment are the same. This can give appearance of dynamic scoping.

Other Languages That Support Lexical Scoping

* Scheme
* Perl Python
* Common 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 global workspace, so everything can be stored on disk because “defining environment” of all functions is the same

## Application: Optimization

Why is this info useful?

* Optimization routines in R like optim, nlm, optimize require a function hose argument is a vector of parameters (e.g. a log-likelihood)
* However, an object function might depend on host of other things besides its paramaters (like *data)*
* When writing software which does optimization, it may be desirable to allow the user to hold certain parameters fixed

## Lexical Scoping Summary

* Objective functions can be “built” which contain all necessary data for evaluation function
* No need to carry around long arguments list - useful for interactive and exploratory work
* Code can be simplified and cleaned up
* Reference: Robert Gentleman and Ross Lhaka (2000). “Lexical Scope and Statistical Computing” JCGS, 9, 491-508

# Coding Standards in R

Coding standards are important to make your code readable.

1. Always right code using text editor, and save as text file (in ASCII). Text editor is least common denominator.
2. Indent your code; different blocks of code should be spaced to right to indicate control flow of program; 8 space indent recommended to help you think about your code
3. Limit the width of your code (80 columns?)
4. Limit the length of individual functions, split into logical function
   1. Nice to have function written on single page of code
   2. When using traceback, profile or debugger, they will tell function where bug occurs; when have multiple functions then it’s easier to go fix it

# Dates and Times in R

R has special way to represent dates and times

* Dates are represented b Date class
* Times are represented by POSIXct or POSIXlt class
* Dates are stored internally as number of days since 1970-01-01
* Times are stored internally as number of seconds since 1970-01-01
* POSIXct is just a very large integer under the hood; when you want to store times in something like a date frame
* POSIXlt is a list underneathand stores a bunch of other useful information like day of week, etc
* weekdays: give the day of the week
* months: give the month name
* quarters: gives the quarter name

## Operations on Date and Times

* You can use mathematical operations on dates and times. Well really just + and -. You can do comparisons to(>=).
* Can’t perform on different classes (Date, POSIXlt, POSIXct).
* Time even keeps track of leap years, leap seconds, daylight savings, time zones

## Date and Time Summary

* Dates and tiems have special classes in R
* Dates use Date class
* Times use POSIXct and POSIXlt class
* Character stirng can be coerced to Date/Time classes using strptime function or the as.Date, asPOSIXlt, as.POSIXct

|  |  |
| --- | --- |
| Command | Description |
| Sys.Date() | returns current date |
| x <- as.Date(“1970-01-01” | declares date x |
| unclass(x)  ## [1] 0 | returns date as stored internally |
| 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] 14.24 | Gives current time (in POSIXct format)  assigns time to POSIX (time in list format)  returns second of p. sec is label for element of list. |
| datestring <- c(“January 10, 2012 10:40”, “December 9, 2011 9:10”)  x <- strptime(datestring, “%B %d, %Y %H:%M”) | takes string and converts to time given provided format  check ?striptime for meanings |
| as.Date(“1969-01-01”) | takes string in date format and stored as Date object |
| weekdays(date)  months(date) | returns day of week from date or time object  returns month from date or time object |
| difftime(Sys.time(), t1, units = 'days') | returns different of time in days from t1 to system time |

# Looping on the Command Line

For and while loops are very useful but not easy when working with interactively on command line. Some functions which implement looping to make life eaiser:

* lapply: Loop over a list and evaluate a function on each element
* sapply: Same as lapply but try to simplify the result
* apply: Apply a function over the margins of an array
* tapply: Apply a function over subsets of a vector
* mapply: Multivariate version of lapply

An auxiliary function split is also useful, particularly in conjunction with lapply

## lapply

* Takes three arguments: 1 a list x; 2 a function or name of function, 3 other arguments. If x is not a list R will try to corerce using as.list
* lapply will always return a list
* lapply and friends make heavy use of *anonymous* functions (functions created on the play)
* sapply will try to simplify the result of lapply if possible
  + if result is list where every element is length 1, then a vector is returned
  + if result is a list where every element if vector of same length (>1), a matrix is returned
  + if neither apply it returns list as with lapply

|  |  |
| --- | --- |
| Command | Description |
| x <- list(a = 1:5, b = rnorm(10))  lapply(x, mean) | creates list of elements - sequence 1 to 5, and 10 random numbers  lapply returns mean for each element in list |
| x <- 1:4  lapply(x,runif) | applies runif to each of x |
| lapply(x, runif, min = 0, max = 10) | applies runif with runif arguments specified |
| x <- list(a = matrix(1:4) |  |
| x <- list(a = matrix(1:4, 2, 2), b = matrix(1:6, 3, 2))  lapply(x,function(elt) elt[,1]) | lapply with anonymous function elt, which takes first column of each matrix in list x |
| x <- list(a = 1:5, b = rnorm(10))  sapply(x,mean) | simplifies result of lapply, returning vector instead of list |
| do call |  |
| lapply(elements, function(elem) elem[2]) | applies anonymous function to each element and returns results |

## apply

apply is used to evaluate a function (often anonymous) over margins of an array

* most often used to apply to rows or column of a matrix
* can be used with general arrays, e.g. taking the average of an array of matrices
* not really faster than writing a loop, but it works on one line

|  |  |
| --- | --- |
| Command | Description |
| x <- matrix(rnorm(200),20,10)  apply(x,2,mean)  apply(x,1,sum) | returns means of each of the 10 columns of matrix x  returns sum of each of the 20 rows of matrix x |
| rowSums = apply(x,1,sum)  rowMeans=apply(x,1,mean)  colSums=apply(x,2,sum)  colMeans=apply(x,2,mean | shortcuts which are equivalent to apply function |
| a <- array(rnorm(2 \* 2 \* 10), c(2,2,10))  apply(a, c(1,2), mean) | collapses third dimension of 3-d array, returning mean for each of first two dimensions |

## mapply

multivariate apply of sorts which applies a function in parallel over a set of arguments

## tapply

Used to apply a function over subsets of a vector. Works on each group based on factor variable (e.g. *gender* - men & women)

|  |  |
| --- | --- |
| Command | Description |
| vapply(variable, function, class/type) | returns a vector or array matching the value or class in third argument |
| mapply(rep,1:4,4:1) | applies function rep to sequence 1:4 and 4:1 |
| x <- c(rnorm(10),runif(10),rnorm(10,1)  f <- gl(3,10) # generates factor levels  tapply(x,f,mean) | takes mean of each group of numbers in x |
| table(array$name) | returns count of records for each unique value in name |
| tapply(flags$animate, flags$landmass, mean) | applies mean to the first variable for each in the second variable |
| ave() | Group Averages Over Level Combinations of Factors |

## split

Takes a vector or other objects and splits it into groups determined by a factor or list of factors.

|  |  |
| --- | --- |
| Command | Description |
| split(vector,factor) | splits vector based on factor into list of vectors |
| s <- split(airquality,airquality$Month) | splits dataframe airquality based on Month variable, s is list of data frames |
| split(x,list(f1,f2), drop = TRUE) | splits vector based on multiple factors, dropping empty factor levels |

# Debugging

Indications that something’s not right

* message: a generics notification/diagnostic message produced by the message function; execution of the function continues
* warning: an indication that something is wrong but not necessarily fatal, execution of the function continues; generated by the warning function
* error: an indication that a fatal problem has occurred; execution stops; produced by the stop function
* condition: a generic concept for indicating that something unexpected can occur; programmers can create their own conditions

How do you know that something is wrong with your function?

* What was your input? How did you call the function?
* What were you expecting? Output, messages, other results?
* What did you get?
* How does what you get differ from what you were expecting?
* Were your expectations correct in the first place?
* Can you reproduce the problem (exactly)?

## Debugging Tools in R

Primary tools for debugging functions in R are:

* traceback: prints out the function call stack after an error occurs, does nothing if there’s no error
* debug: flags a function for “debug” mode which allows you to step through execution of a function one line at a time
* browser: suspends the execution of a function wherever it is called and puts the function in debug mode
* trace: allows you to insert debugging code into a function at specific places
* recover: allows you to modify the error behaviour so that you can browser the function call stack

These are interactive tools designed to allow you to pick through a function. There’s also a more blunt technique of inserting print/cat statements in the function.

## Summary

* Three main problem/conditions: message, warning, error
* When analysing function with problem, make sure you can reproduce problem, clearly state expectations and how output differs from your expectations
* Interactive debugging tools traceback, debug, browswer, trace, and recoever can be used to find problematic code in functions
* Debugging tools are not a substitute for thinking

|  |  |
| --- | --- |
| Command | Description |
| traceback() | shows where previous error occurred using call stack. must be called exactly after where error occurred. |
| browser(function)  n | opens browser in function environment  executres next line in function |
| options(error = recover) | R will go into recover whenever error occurs during current session |
| Browse[2]> undebug(ls)  Browse[2]> Q | get out of debug mode |
| stopifnot() | equivalent of assert in python |

# str Function

* The most important function in all of R
* A diagnostic function and alternative to summary()
* it is especially well suited to compactly display the (abbreviated) contents of (possibly nested) lists
* Roughly one line per basic object
* Goal: **What’s in this object**

|  |  |
| --- | --- |
| Command | Description |
| head(df,nrows) | displays first number of rows (nrows) of a data frame df |
| tail(df,nrows) | displays last number of rows (nrows) of a data frame df |
| summary(df) | displays distribute of data frame for each categorical/factor variable; min/mean/max and quantiles for numeric variables, and count of NA’s for each |
| str(df) | displays structure of data frame |

# Simulation

functions for probability distributions in R

* rnorm: generate random Normal variates with a given mean and standard deviation
* dnorm: evaluate the Normal probability density (with a given mean/SD) at a point (or vector of points)
* pnorm: evaluate the cumulative distribution function for a Normal distribution
* rpois: generate random Poisson variates with a given rate

## Generating Random Numbers

such function usually have four functions associated with them. These are prefixed with:

* d for density
* r for random number generation
* p for cumulative distribution
* q for quantile function

Setting the random number seed with set.seed ensure reproducibility. Always set this when conducting a simulation.

## Random Sampling

The sample function draws randomly from a specified set of (scalar) objects allowing you to sample from arbitrary distributions

## Simulation Summary

* drawing samples from specific probability distributions can be done with r\* functions
* Standard distributions are built int: Normal, Poisson, Binomial, Exponential, Gamma
* The sample function can be used to draw random samples from arbitrary vectors
* S the random number generator seed via set.seed is critical for reproducibility

|  |  |
| --- | --- |
| Command | Description |
| sample(dataset, 4, replace = TRUE) | randomly samples 4 values from dataset, allowing replacements |
| sample(dataset, 10) | randomly samples 4 values from dataset, forbidding replacements |
| flips <- sample(c(0,1), 100 replace = TRUE, prob = c(0.3,0.7)) | assigns 100 sample of 0 or 1 with 30% probability 0 will be chosen, and 70% probability 1 with be chosen |
| flips2 <- rbinom(100, size = 1, prob = 0.7) | random binomial distribution |
| rnorm(10,mean = 100, sd = 25) | generates 10 numbers from random normal distribution with mean of 100 and standard deviation of 25 |
| replicate(100, rpois(5,10)) | generates 5 numbers from ramdom Poisson distribution repeated 100 times and stored in a matrix of 5 rows and 100 columns |
| set.seed(n) | sets seed for random number generation |
| hist(vector) | generates histogram plot of values in a vector |

# R Profiling

* Profiling is a systematic way to examine how much time is spent in different parts of a program
* useful when trying to optimize code
* often code runs fine once, but what if you have to put it in a loop for 1000 iterations
* Profiling is better than guessing

## Optimizing Your Code

* Getting biggest impact on speeding up code depends on knowing where the code spends most of its time
* This cannot be done without performance analysis or profiling
* Design first, then optimize
* Remember: premature optimization is the root of all evil
* Measure (collect data), don’t guess
* If you’re going to be a scientist, you need to apply the same principles here

## system.time()

* takes arbitrary R expression as input (wrapped in curly braces) and returns amount of time to evaluate expression
* Computes time (in sec) needed to execute an expression
* If there’s an error, gives time until error ocurred’
* Returns an object of class proc\_time:
  + user time: time charged to CPU for expression
  + elapsed time: wall ‘clock’ time
* Usually, user time and elapses time are relatively close, for straight computing tasks
* Elapsed time may be greater than user time if CPU spends a lot of time waiting around
* Elapsed time may be smaller than user time if your machine has multiple cores/processors (and is capable of using them)
  + Multi-threaded BLAS libraries (vecLib/Accelerate, ATLAS< ACML, MKL)
  + Parallel processing via the parallel package
* Using system.time() allows you to test certain functions or code blocks to see if they are taking excessive amounts of time
* Assumes you already know where the problem is and can call system.time() on it
* What if you don’t know where to start?

## RProf()

* The RProf() function starts the profiler in R
* summaryRprof() function summarizes the output from Rprof()
* DO NOT USE system.time() and Rprof() together
* Rprof() keeps track of function call stack at regularly samples intervals and tabulates how much time is spent on each function
* Default sampling interval is 0.02 seconds
* Note: if your code runs very quickly, profiler is not useful

## Using summaryRprof()

* tabulates the R profiler output and calculates how much time is spend in which function
* two methods for normalizing the data
  + “by.total” divides the time spent in each function by total run time
  + “by.self” does same but first subtracts out time spent in functions above call stack
  + by.self is more accurate of which functions are truly taking up most amount of time

## Summary

* Rprof() runs the profiler for performance of analysis of R code
* summaryRprof() summarizes output of Rprof() and gives percent time spent ine ach function, with two types of normalization - by.self is best
* Good to break your code into functions so that profiler can give useful information about where time is being spent
* C or Fortran code not profiled

# Base Graphics

|  |  |
| --- | --- |
| Command | Description |
| plot(data) | displays x and y axis in scatter plot for two variables, the first column plotted as x and second column plotted as y |
| plot(x = cars$speed, y = cars$dist) | plot with the x and y variables explicitly defined |
| plot(x = cars$speed, y = cars$dist , xlab = "Speed", ylab = "Stopping Distance") | plot with the x and y axes labeled |
| boxplot(formula = mpg ~ cyl, data = mtcars) | creates boxplot with cyl on x-axis and mpg on y-axis |
| hist(mtcars$mpg) | creates histogram of mpg variable |