**R Programming**

**Module 1: Getting Started and R Nuts and Bolts**

|  |  |
| --- | --- |
| 1. Overview and History of R  2. R Console Input and Evaluation  3. Data Types  3.1 R Objects and Attributes  3.2 Vectors and Lists  3.3 Factors  3.4 Missing values  3.5 Data Frames  3.6 Names Attribute  3.7 Summary  4. Reading Data  4.1 Reading Tabular Data  4.2 Reading Large Table  4.3 Textual Data Formats  4.4 Connections: Interfaces to the Outside World  5. Subsetting  5.1 Basics  5.2 Lists  5.3 Matrices  5.4 Partial Matching  5.5 Removing Missing Values  6. Vectorized Operations | * R: a dialect of the S language * Features of R:   Runs on almost any standard computing platform/OS  Frequent 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  It's free!   * Drawbacks of R   Based on 40 year old technology.  Little built in support for dynamic or 3-D graphics  Functionality is based on consumer demand and user contributions.  Objects must generally be stored in physical memory  Not ideal for all possible situations   * Design of the R System   2 parts:  1. The “base” R system that you download from CRAN  2. Everything else  R functionality is divided into a number of packages:  1. The “base” R system: is required to run R and contains the most fundamental functions  2. Other packages contained in the “base” system: utils, stats, datasets, graphics, grDevices, grid, methods, tools, parallel, compiler, splines, tcltk, stats4  3. “Recommend” Packages: boot, class, cluster, codetools, foreign, KernSmooth, lattice, mgcv, nlme, rpart, survival, MASS, spatial, nnet, Matrix  Many other packages available: CRAN, Bioconductor   * Some R Resources   Available from CRAN (http://cran.r-project.org)  An Introduction to R  Writing R Extensions  R Data Import/Export  R Installation and Administration (mostly for building R from sources)  R Internals (not for the faint of heart)   * Entering Input:   At the R prompt we type expressions.  The <- symbol is the assignment operator.  The # character indicates a comment.  > x <- 1  > print(x)  [1] 1  > x  [1] 1  > msg <- "hello"  > x <- ## Incomplete expression   * Evaluation: When a complete expression is entered at the prompt, it is evaluated and the result of the evaluated expression is returned. The result may be auto-printed.   > x <- 5 ## nothing printed  > x ## auto-printing occurs  [1] 5  > print(x) ## explicit printing  [1] 5  The [1] indicates that x is a vector and 5 is the first element.  > x <- 1:20  > x  [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15  [16] 16 17 18 19 20  The : operator is used to create integer sequences.   * 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:   * can only contain objects of the same class * Exception: list, which is represented as a vector but can contain objects of different classes   Empty vectors can be created with the vector() function.   * Numbers:   Numbers in R a generally treated as numeric objects  If you explicitly want an integer, you need to specify the L suffix  i.e. Entering 1 gives you a numeric object; entering 1L explicitly gives you an integer.  Special numbers:   * Inf: represents infinity * NaN: represents an undefined value (“not a number”) * 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.   * Creating Vectors: * c() function: create vector 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   * vector() function   > x <- vector("numeric", length = 10)  > x  [1] 0 0 0 0 0 0 0 0 0 0  Mixing Objects: When different objects are mixed in a vector, coercion occurs so that every element in the vector is of the same class.  > y <- c(1.7, "a") ## character  > y <- c(TRUE, 2) ## numeric  > y <- c("a", TRUE) ## character  Explicit Coercion: Objects can be explicitly coerced from one class to another using the as.\* functions. Nonsensical coercion results in NAs.  > 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"   * Matrices: 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  cbind-ing and rbind-ing: 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   * Lists: a special type of vector that can contain elements of different classes.   > x <- list(1, "a", TRUE, 1 + 4i)  > x  [[1]]  [1] 1  [[2]]  [1] "a"  [[3]]  [1] TRUE  [[4]]  [1] 1+4i   * Factors:   used to represent categorical data  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   > 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. (If not set, level order is based on alphabetical order.  > x <- factor(c("yes", "yes", "no", "yes", "no"), levels = c("yes", "no"))  > x  [1] yes yes no yes no  Levels: yes no   * 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: 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()   > 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   * R objects can also have names, which is very useful for writing readable code and self-describing objects. Names() function   > 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  Matrices can also have names. dimnames() <- list(c(Raw names), c(Column names))  > 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   * Data Types * atomic classes: numeric, logical, character, integer, complex \ * vectors, lists * factors * missing values * data frames * names * Reading Data:   There are a few principal functions reading data into R.   * + read.table, read.csv, for reading tabular data   + readLines, for reading lines of a text file   + source, for reading in R code files (inverse of dump)   + dget, for reading in R code files (inverse of dput)   + load, for reading in saved workspaces   + unserialize, for reading single R objects in binary form * Writing Data   There are analogous functions for writing data to files   * + write.table   + writeLines   + dump   + dput   + save   + serialize * Reading Data Files with read.table   The read.table function is one of the most commonly used functions for reading data. It has a few important arguments:   * + file, the name of a file, or a connection   + header, logical indicating if the file has a header line   + sep, a string indicating how the columns are separated   + colClasses, a character vector indicating the class of each column in the dataset   + nrows, the number of rows in the dataset   + comment.char, a character string indicating the comment character   + skip, the number of lines to skip from the beginning   + stringsAsFactors, should character variables be coded as factors?   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 * Read the help page for read.table, which contains many hints * Make a rough calculation of the memory required to store your dataset * 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.   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. * Know The System:   How much memory is available?  What other applications are in use?  Are there other users logged into the same system?  What operating system?  Is the OS 32 or 64 bit?   * Calculating Memory Requirements:   I have a data frame with 1,500,000 rows and 120 columns, all of which are numeric data. Roughly, how much memory is required to store this data frame?  1,500,000 × 120 × 8 bytes/numeric  = 1440000000 bytes  = 1440000000 / bytes/MB  = 1,373.29 MB  = 1.34 GB   * Textual Formats:   + 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 * dput-ting R Objects:   Another way to pass data around is by deparsing the R object with dput and reading it back in using dget.  > y <- data.frame(a = 1, b = "a")  > dput(y)  structure(list(a = 1,  b = structure(1L, .Label = "a",  class = "factor")),  .Names = c("a", "b"), row.names = c(NA, -1L),  class = "data.frame")  > dput(y, file = "y.R")  > new.y <- dget("y.R")  > new.y  a b  1 1 a   * Dumping R Objects Dumping R Object   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"   * 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)   * Connections:   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   * 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"  > 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"  First type: numeric index  Second type: logical index   * [ or [[ or $   > 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   * extract multiple subset: can only use [   > 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   * 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   * Matrices can be subsetted in the usual way with (row,column) type indices.   > x <- matrix(1:6, 2, 3)  > x[1, 2]  [1] 3  > x[2, 1]  [1] 2  Indices can also be missing.  > 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 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   * A common task * logical function: is.na(), complete.cases()   complete.cases() is useful in removing all missing values  > x <- c(1, 2, NA, 4, NA, 5)  > bad <- is.na(x)  > x[!bad]  [1] 1 2 4 5     * Many operations in R are vectorized making code more efficient, concise, and easier to read.   + \* - / etc. : element-wise multiplication  % +\*-/ % : true matrix multiplication  > 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 |

**Module 2: Programming with R**

|  |  |
| --- | --- |
| 1. Control Structures  1.1 Control Structures: if  1.2 Control Structures: For loops  1.3 Control Structures: While loops  1.4 Control Structures: Repeat, Next, Break  1.5 Summary  2. Functions  2.1 Function Arguments  2.2 Lazy Evaluation  2.3 The “…” Argument  3. Scoping Rules  3.1 Binding Values to Symbol  3.2 Scoping Rules  4. Coding Standards  5. Dates and Times | * 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 expressions.   * if(<condition>) {   ## do something  } else {  ## do something else  }   * if(<condition1>) {   ## do something  } else if(<condition2>) {  ## do something different  } else {  ## do something different  }   * Of course, the else clause is not necessary.   if(<condition1>) {  }  if(<condition2>) {  }   * 2 types:   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  }   * 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)) { # seq\_along(x) generate an integer sequence 0, 1, 2… same length as 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. However: Be careful with nesting though. Nesting beyond 2–3 levels is often very difficult to read/understand.   x <- matrix(1:6, 2, 3)  for(i in seq\_len(nrow(x))) { #seq\_len(a) generate an integer sequence 0, 1, 2… same length as a  for(j in seq\_len(ncol(x))) {  print(x[i, j])  }  }   * 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.   While loops can potentially result in infinite loops if not written properly. Use with care!  count <- 0  while(count < 10) {  print(count)  count <- count + 1  }   * Sometimes there will be more than one condition in the test. (Use logic characters) Conditions are always evaluated from left to right.   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  }  }   * 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() #Note: this is not a real function  if(abs(x1 - x0) < tol) {  break  } else {  x0 <- x1  }  }  The loop 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  return signals that a function should exit and return a given value  for(i in 1:100) {  if(i <= 20) {  ## Skip the first 20 iterations  next  }  ## Do something here  }   * Control structures like if, while, and for allow you to control the flow of an R program * Infinite loops should generally be avoided, even if they are theoretically correct. * Control structures mentioned here are primarily useful for writing programs; for command-line interactive work, the \*apply functions are more useful. * 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. * 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.   The following calls to sd are all equivalent. Even though it’s legal, it is not recommend messing around with the order of the arguments too much, since it can lead to some confusion.  > 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)   * 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.   * 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   * In addition to not specifying a default value, you can also set an argument value to NULL.   f <- function(a, b = 1, c = 2, d = NULL) {  }   * 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 indicate a variable number of arguments that are usually passed on to other functions. 2 usages: * ... 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.   > 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)  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 :"   * 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 a R object, the order is roughly: * Search the global environment for a symbol name matching the one requested. * 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"  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.   * The scoping rules for R:   + 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: the values of free variables are searched for in the environment in which the function was defined.   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. * 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. * Why does lexical scoping matters?   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  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  What’s in a function’s environment? ls()  > 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)   * 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. * 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.   * 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. * 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 * Indenting   + Indenting improves readability   + Fixing line length (80 columns) prevents lots of nesting and very long functions   + Suggested: Indents of 4 spaces at minimum; 8 spaces ideal * 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 * Times are stored internally as the number of seconds since 1970-01-01 * Dates   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 * 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 9:10")  x <- strptime(datestring, "%B %d, %Y %H:%M")  ## [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   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   * 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 function or the as.Date, as.POSIXlt, or as.POSIXct |

**Module 3: Loop Functions and Debugging**

|  |  |
| --- | --- |
| **1. Loop functions**    1.1 lapply and sapply  1.2 apply  1.3 mapply  1.4 tapply  1.5 split  2. Debugging Tools  2.1 Diagnosing the Problem  2.2 Basic Tools  Summary | * Looping in the command line:   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 the name of a function) FUN; (3) other arguments via its ... argument. If X is not a list, it will be coerced to a list using as.list   ## function (X, FUN, ...)  ## {  ## FUN <- match.fun(FUN)  ## if (!is.vector(X) || is.object(X))  ## X <- as.list(X)  ## .Internal(lapply(X, FUN))  ## }  ## <bytecode: 0x7ff7a1951c00>  ## <environment: namespace:base>   * lapply always returns a list, regardless of the class of the input.   x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1), d = rnorm(100, 5))  lapply(x, mean)  ## $a  ## [1] 2.5  ## $b  ## [1] 0.5261  ## $c  ## [1] 1.421  ## $d  ## [1] 4.927    > x <- 1:4  > lapply(x, runif)  [[1]]  [1] 0.2675082  [[2]]  [1] 0.2186453 0.5167968  [[3]]  [1] 0.2689506 0.1811683 0.5185761  [[4]]  [1] 0.5627829 0.1291569 0.2563676 0.7179353    > x <- 1:4  > lapply(x, runif, min = 0, max = 10)  [[1]]  [1] 3.302142  [[2]]  [1] 6.848960 7.195282  [[3]]  [1] 3.5031416 0.8465707 9.7421014  [[4]]  [1] 1.195114 3.594027 2.930794 2.766946   * lapply and friends make heavy use of anonymous functions.   X <- list(a = matrix(1:4, 2, 2), b = matrix(1:6, 3, 2)  lapply(X, function(let), let[,1]  $a  [1] 1 2  $b  [1] 1 2 3   * sapply will try to simplify the result of lapply if possible.   If the result is a list where every element is length 1, then a vector is returned  If the result is a list where every element is a vector of the same length (> 1), a matrix is returned.  If it can’t figure things out, a list is returned  > x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1), d = rnorm(100, 5))  > lapply(x, mean)  $a  [1] 2.5  $b  [1] 0.06082667  $c  [1] 1.467083  $d  [1] 5.074749  > sapply(x, mean)  a b c d  2.50000000 0.06082667 1.46708277 5.07474950   * apply is used to evaluate a function (often an anonymous one) over the margins of an array.   It is most often used to apply a function to the rows or columns of a matrix  It can be used with general arrays, e.g. taking the average of an array of matrices  It is not really faster than writing a loop, but it works in one line!   * function (X, MARGIN, FUN, ...)   X is an array  MARGIN is an integer vector indicating which margins should be “retained”.  FUN is a function to be applied  ... is for other arguments to be passed to FUN  > x <- matrix(rnorm(200), 20, 10)  > apply(x, 2, mean) #reserve column and collapse row  [1] 0.04868268 0.35743615 -0.09104379  [4] -0.05381370 -0.16552070 -0.18192493  [7] 0.10285727 0.36519270 0.14898850  [10] 0.26767260  > apply(x, 1, sum) #reserve row and collapse column  [1] -1.94843314 2.60601195 1.51772391  [4] -2.80386816 3.73728682 -1.69371360  [7] 0.02359932 3.91874808 -2.39902859  [10] 0.48685925 -1.77576824 -3.34016277  [13] 4.04101009 0.46515429 1.83687755  [16] 4.36744690 2.21993789 2.60983764  [19] -1.48607630 3.58709251    > a <- array(rnorm(2 \* 2 \* 10), c(2, 2, 10))  > apply(a, c(1, 2), mean)  [,1] [,2]  [1,] -0.2353245 -0.03980211  [2,] -0.3339748 0.04364908  > rowMeans(a, dims = 2)  [,1] [,2]  [1,] -0.2353245 -0.03980211  [2,] -0.3339748 0.04364908   * col/row sums and means   For sums and means of matrix dimensions, we have some shortcuts.  The shortcut functions are much faster, but you won’t notice unless you’re using a large matrix.  rowSums = apply(x, 1, sum)  rowMeans = apply(x, 1, mean)  colSums = apply(x, 2, sum)  colMeans = apply(x, 2, mean)   * mapply is a multivariate apply of sorts which applies a function in parallel over a set of arguments.   function (FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE, USE.NAMES = TRUE)  FUN is a function to apply  ... contains arguments to apply over  MoreArgs is a list of other arguments to FUN.  SIMPLIFY indicates whether the result should be simplified  The following is tedious to type  > list(rep(1, 4), rep(2, 3), rep(3, 2), rep(4, 1))  Instead we can do  > mapply(rep, 1:4, 4:1)  [[1]]  [1] 1 1 1 1  [[2]]  [1] 2 2 2  [[3]]  [1] 3 3  [[4]]  [1] 4  > noise <- function(n, mean, sd) {  + rnorm(n, mean, sd)  + }  > noise(5, 1, 2)  [1] 2.4831198 2.4790100 0.4855190 -1.2117759  [5] -0.2743532  > noise(1:5, 1:5, 2) #This is not what we want  [1] -4.2128648 -0.3989266 4.2507057 1.1572738  [5] 3.7413584  Instead we do  > mapply(noise, 1:5, 1:5, 2)  [[1]]  [1] 1.037658  [[2]]  [1] 0.7113482 2.7555797  [[3]]  [1] 2.769527 1.643568 4.597882  [[4]]  [1] 4.476741 5.658653 3.962813 1.204284  [[5]]  [1] 4.797123 6.314616 4.969892 6.530432 6.723254  Which is the same as  list(noise(1, 1, 2), noise(2, 2, 2), noise(3, 3, 2), noise(4, 4, 2), noise(5, 5, 2))   * tapply is used to apply a function over subsets of a vector.   function (X, INDEX, FUN = NULL, ..., simplify = TRUE)  X is a vector  INDEX is a factor or a list of factors (or else they are coerced to factors)  FUN is a function to be applied  ... contains other arguments to be passed FUN  simplify, should we simplify the result?   * ·Take group means   > x <- c(rnorm(10), runif(10), rnorm(10, 1))  > f <- gl(3, 10) #3 levels, each level repeats 10 times  > f  [1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3  [24] 3 3 3 3 3 3 3  Levels: 1 2 3  > tapply(x, f, mean)  1 2 3  0.1144464 0.5163468 1.2463678   * Take group means without simplification   > tapply(x, f, mean, simplify = FALSE)  $‘1‘  [1] 0.1144464  $‘2‘  [1] 0.5163468  $‘3‘  [1] 1.246368   * Find group ranges   > tapply(x, f, range)  $‘1‘  [1] -1.097309 2.694970  $‘2‘  [1] 0.09479023 0.79107293  $‘3‘  [1] 0.4717443 2.5887025   * split takes a vector or other objects and splits it into groups determined by a factor or list of factors.   function (x, f, drop = FALSE, ...)  x is a vector (or list) or data frame  f is a factor (or coerced to one) or a list of factors  drop indicates whether empty factors levels should be dropped    A common idiom is split followed by an lapply.  > lapply(split(x, f), mean)  $‘1‘  [1] 0.1144464  $‘2‘  [1] 0.5163468  $‘3‘  [1] 1.246368   * Splitting a dataframe          * Splitting on More than One Level   > x <- rnorm(10)  > f1 <- gl(2, 5)  > f2 <- gl(5, 2)  > f1  [1] 1 1 1 1 1 2 2 2 2 2  Levels: 1 2  > f2  [1] 1 1 2 2 3 3 4 4 5 5  Levels: 1 2 3 4 5  > interaction(f1, f2)  [1] 1.1 1.1 1.2 1.2 1.3 2.3 2.4 2.4 2.5 2.5  10 Levels: 1.1 2.1 1.2 2.2 1.3 2.3 1.4 ... 2.5  Interactions can create empty levels.  > str(split(x, list(f1, f2)))  List of 10  $ 1.1: num [1:2] -0.378 0.445  $ 2.1: num(0)  $ 1.2: num [1:2] 1.4066 0.0166  $ 2.2: num(0)  $ 1.3: num -0.355  $ 2.3: num 0.315  $ 1.4: num(0)  $ 2.4: num [1:2] -0.907 0.723  $ 1.5: num(0)  $ 2.5: num [1:2] 0.732 0.360  Empty levels can be dropped.  > str(split(x, list(f1, f2), drop = TRUE))  List of 6  $ 1.1: num [1:2] -0.378 0.445  $ 1.2: num [1:2] 1.4066 0.0166  $ 1.3: num -0.355  $ 2.3: num 0.315  $ 2.4: num [1:2] -0.907 0.723  $ 2.5: num [1:2] 0.732 0.360   * Something’s Wrong!   Indications that something’s not right  message: A generic 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  log(-1)  ## Warning: NaNs produced  ## [1] NaN  printmessage <- function(x) {  if (x > 0)  print("x is greater than zero") else print("x is less than or equal to zero")  invisible(x)  }  printmessage(NA)  ## Error: missing value where TRUE/FALSE needed  Revise:  printmessage2 <- function(x) {  if(is.na(x))  print("x is a missing value!")  else if(x > 0)  print("x is greater than zero")  else  print("x is less than or equal to zero")  invisible(x)  }  x <- log(-1)  ## Warning: NaNs produced  printmessage2(x)  ## [1] "x is a missing value!"   * 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)? * The 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 a specific places * recover: allows you to modify the error behavior so that you can browse the function call stack   These are interactive tools specifically designed to allow you to pick through a function. There’s also the more blunt technique of inserting print/cat statements in the function.   * Traceback   > mean(x)  Error in mean(x) : object 'x' not found  > traceback()  1: mean(x)  > lm(y ~ x)  Error in eval(expr, envir, enclos) : object ’y’ not found  > traceback()  7: eval(expr, envir, enclos)  6: eval(predvars, data, env)  5: model.frame.default(formula = y ~ x, drop.unused.levels = TRUE)  4: model.frame(formula = y ~ x, drop.unused.levels = TRUE)  3: eval(expr, envir, enclos)  2: eval(mf, parent.frame())  1: lm(y ~ x)   * Debug   > debug(lm)  > lm(y ~ x)  debugging in: lm(y ~ x)  debug: {  ret.x <- x  ret.y <- y  cl <- match.call()  ...  if (!qr)  z$qr <- NULL  z  }  Browse[2]>  Browse[2]> n  debug: ret.x <- x  Browse[2]> n  debug: ret.y <- y  Browse[2]> n  debug: cl <- match.call()  Browse[2]> n  debug: mf <- match.call(expand.dots = FALSE)  Browse[2]> n  debug: m <- match(c("formula", "data", "subset", "weights", "na.action", "offset"), names(mf), 0L)   * Recover   > options(error = recover)  > read.csv("nosuchfile")  Error in file(file, "rt") : cannot open the connection  In addition: Warning message:  In file(file, "rt") :  cannot open file ’nosuchfile’: No such file or directory  Enter a frame number, or 0 to exit  1: read.csv("nosuchfile")  2: read.table(file = file, header = header, sep = sep, quote = quote, dec =  3: file(file, "rt")  Selection:   * There are three main indications of a problem/condition: message, warning, error   - only an error is fatal   * When analyzing a function with a problem, make sure you can reproduce the problem, clearly state your expectations and how the output differs from your expectation * Interactive debugging tools traceback, debug, browser, trace, and recover can be used to find problematic code in functions * Debugging tools are not a substitute for thinking! |

**Module 4: Simulation & Profiling**

|  |  |
| --- | --- |
| 1. The str Function  2. Simulation  2.1 Generating Random Numbers  2.2 Simulating a Linear Model  2.3 Random Sampling  2.4 Summary  3. Profiling R Code  3.1 Using system.time()  3.2 The R Profiler  3.3 Summary | * Compactly display the internal structure of an R object   A diagnostic function and an alternative to ‘summary’  It is especially well suited to compactly display the (abbreviated)contents of (possibly nested) lists.  Roughly one line per basic object  “What’s in this object?”   * 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   * Probability distribution functions usually have four functions associated with them. The functions are prefixed with a   d for density  r for random number generation  p for cumulative distribution  q for quantile function   * Working with the Normal distributions requires using these four functions   dnorm(x, mean = 0, sd = 1, log = FALSE)  pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)  qnorm(p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)  rnorm(n, mean = 0, sd = 1)  If Φ is the cumulative distribution function for a standard Normal distribution, then pnorm(q) =Φ(q) and qnorm(p) =Φ-1(p).   * Setting the random number seed with set.seed ensures reproducibility   > set.seed(1)  > rnorm(5)  [1] -0.6264538 0.1836433 -0.8356286 1.5952808  [5] 0.3295078  > rnorm(5)  [1] -0.8204684 0.4874291 0.7383247 0.5757814  [5] -0.3053884  > set.seed(1) #go back to the start of last rnorm()  > rnorm(5)  [1] -0.6264538 0.1836433 -0.8356286 1.5952808  [5] 0.3295078  Always set the random number seed when conducting a simulation!   * Generating Poisson data   > rpois(10, 1)  [1] 3 1 0 1 0 0 1 0 1 1  > rpois(10, 2)  [1] 6 2 2 1 3 2 2 1 1 2  > rpois(10, 20)  [1] 20 11 21 20 20 21 17 15 24 20  > ppois(2, 2) ## Cumulative distribution  [1] 0.6766764 ## Pr(x <= 2)  > ppois(4, 2)  [1] 0.947347 ## Pr(x <= 4)  > ppois(6, 2)  [1] 0.9954662 ## Pr(x <= 6)   * Generating Random Numbers From a Linear Model   Suppose we want to simulate from the following linear model  y = β0 + β1x + ε  where ε ∼ N(0, 2 ) . Assume x ∼ N(0, 1 ), β0 = 0.5 and β1 = 2.  > set.seed(20)  > x <- rnorm(100)  > e <- rnorm(100, 0, 2)  > y <- 0.5 + 2 \* x + e  > summary(y)  Min. 1st Qu. Median  -6.4080 -1.5400 0.6789 0.6893 2.9300 6.5050  > plot(x, y)    What if x is binary?  > set.seed(10)  > x <- rbinom(100, 1, 0.5)  > e <- rnorm(100, 0, 2)  > y <- 0.5 + 2 \* x + e  > summary(y)  Min. 1st Qu. Median  -3.4940 -0.1409 1.5770 1.4320 2.8400 6.9410  > plot(x, y)    Suppose we want to simulate from a Poisson model where  Y ~ Poisson(μ)  log μ = β0 + β1x  andβ0 = 0.5 and β1 = 0.3. We need to use the rpois function for this  > set.seed(1)  > x <- rnorm(100)  > log.mu <- 0.5 + 0.3 \* x  > y <- rpois(100, exp(log.mu))  > summary(y)  Min. 1st Qu. Median Mean 3rd Qu. Max.  0.00 1.00 1.00 1.55 2.00 6.00  > plot(x, y)     * The sample function draws randomly from a specified set of (scalar) objects allowing you to sample from arbitrary distributions.   > set.seed(1)  > sample(1:10, 4)  [1] 3 4 5 7  > sample(1:10, 4)  [1] 3 9 8 5  > sample(letters, 5)  [1] "q" "b" "e" "x" "p"  > sample(1:10) ## permutation  [1] 4 710 6 9 2 8 3 1 5  > sample(1:10)  [1] 2 3 4 1 9 5 10 8 6 7  > sample(1:10, replace = TRUE) ## Sample w/replacement  [1] 2 9 7 8 2 8 5 9 7 8   * Drawing samples from specific probability distributions can be done with r\* functions * Standard distributions are built in: Normal, Poisson, Binomial, Exponential, Gamma, etc. * The sample function can be used to draw random samples from arbitrary vectors * Setting the random number generator seed via set.seed is critical for reproducibility * Why is my code so slow?   Profiling is a systematic way to examine how much time is spend in different parts of a program  Useful when trying to optimize your code  Often code runs fine once, but what if you have to put it in a loop for 1,000 iterations? Is it still fast enough?  Profiling is better than guessing   * On 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   * General principles of optimization   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 scientist, you need to apply the same principles here!   * Takes an arbitrary R expression as input (can be wrapped in curly braces) and returns the amount of time taken to evaluate the expression * Computes the time (in seconds) needed to execute an expression   - If there’s an error, gives time until the error occurred   * Returns an object of class proc\_time   - user time: time charged to the CPU(s) for this expression  - elapsed time: "wall clock" time  Usually, the user time and elapsed time are relatively close, for straight computing tasks  Elapsed time may be greater than user time if the CPU spends a lot of time waiting around  Elapsed time may be smaller than the 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   * Beyond system.time()   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?   * The Rprof() function starts the profiler in R   - R must be compiled with profiler support (but this is usually the case)  The summaryRprof() function summarized the output from Rprof() (otherwise it is not readable)  DO NOT use system.time() and Rprof() together or you will be sad  Rprof() keeps track of the function call stack at regularly sampled intervals and tabulates how much time is spend in each function  Default sampling interval is 0.02 seconds  NOTE: If your code runs very quickly, the profiler is not useful, but then you probably don't need it in that case·  R Profiler Raw Output:     * Using summaryRprof()   The summaryRprof() function tabulates the R profiler output and calculates how much time is spend in which function  There are two methods for normalizing the data  "by.total" divides the time spend in each function by the total run time  "by.self" does the same but first subtracts out time spent in functions above in the call stack      summaryRprof() Output:     * Rprof() runs the profiler for performance of analysis of R code * summaryRprof() summarizes the output of Rprof() and gives percent of time spent in each function (with two types of normalization) * Good to break your code into functions so that the profiler can give useful information about where time is being spent * C or Fortran code is not profiled |