R Programming

Introduction

- This chapter will discuss the basic building blocks of R, basic data types, writing R scripts, profiling code and debugging

Overview and History of R

- Brief history of how the language evolved (not pertinent just background information)
- Designed from the outset to be an interactive environment where the user could easily perform basic work with data, but as their needs became more sophisticated, they could slide into gradually into programming
- Benefits from frequent updates and bug fixes
- Divided into modular packages so you only have to download what you need
- Graphics capabilities are superior to most other statistic packages
- High active user community who can help with issues and frequently release their own packages which enhance the functionality of R or simplify certain processes
- However, there is little support for dynamic or 3-D graphics, functionality is based on demand so if not enough people care about an issue it won't be solved, objects need to be stored in physical memory and there are certain situations where it struggles

Getting Help

- Very similar content to the Getting Help video in Chapter 1: The Data Scientist's Toolbox
- Check spelling mistakes in code
- Search the web and forum archives; you can try pasting any error messages (e.g. Stack Overflow)
- Read the manual or FAQs
- Experiment and see what happens
- Ask a skilled friend/colleague
- Post to a forum (ensuring it hasn't already been asked)
- When posting to a forum ensure you include:
    - Steps to recreate the problem including just enough example data as is necessary to illustrate the issue (don't swamp people with unnecessary data)
    - What you already tried
    - What happened and what you expected to happen
    - What OS and version of software/packages you are using
    - Additional context
    - A succinct title which describes the specific issue you are having
- Follow up with the solution so that others having a similar issue can benefit

RConsole Input and Evaluation

- <- is the assignment operator and assigns a value to a variable via variable <- value
- # is used to write comments
- The operator : is used to create integer sequences

Data Types – R Objects and Attributes

- R has 5 basic atomic classes of objects:
    - Character
    - Numeric (real numbers)
    - Integer
    - Complex
    - Logical (True/False)
- A vector can only contain objects of the same class with the exception of a list which is represented as a vector but can contain objects of different classes
- Use vector() to create an empty vector
- Inf represents infinity and can be used in calculations (e.g. 1/0 is Inf)
- NaN ("Not a Number") represents an undefined or missing value
- Objects can have attributes (accessed via attributes()):
    - Names, dimnames
    - Dimensions
    - Class
    - Length
    - Other user-defined attributes

Data Types – Vectors and Lists

- Vectors are created using c() (c for concatenate)
- You can specify the class and length of blank vectors with vector("class", length = n)
- When you mix object classes within a vector coercion occurs; this is where every element is changed to it's representation in the same class e.g.
    - c(1.7, "a") - character with 1.7 coered to string
    - c(TRUE, 2) - numeric with TRUE coerced to 1
    - C("a", TRUE) - character with TRUE coerced to string
- You can explicitly coerce from one class to another via as.class() e.g. as.numeric or as.character
- Explicit coercion isn't always possible e.g. as.numeric(c("a", "b")) returns NA plus a warning
- A list is similar to a vector but can contain objects of different classes and the objects are indexed by double brackets

Data Types – Matrices

- Matrices are like vectors with the additional dimension attribute

- Empty matrix is created via matrix(nrow=n, ncol=m)
- Matrices are contructed column-wise starting in the upper left and ending in the lower right
- Matrices can be created by combining rows or columns via the cbind() and rbind() functions
-

Data Types – Factors

- Store categorical data which can be ranked (e.g. small/medium/large) or unranked (e.g. male/female)
- Factor variables are useful since they are self-describing; i.e. male /female is more descriptive than 1/0
- Created using factor() and when printed will show the levels
- Calling table() on a factor variable will show a tables wit the frequency of each level
- Unclass() will strip out the classes from a factor variable
- The levels argument of factor() is used to order the levels with the first level designated as the baseline

Data Types – Missing Values

- is.na() and is.nan(0 are sued to test for NA and NaN
- NA values also have a class e.g. integer

Data Types – Data Frames

- Data frames are used to store tabular data
- Represented as a list where every element has the same length
- Data frames have a special attribute called row.names
- Created by read.table() or read.csv()

Data Types – Names Attribute

- R objects can be assigned names via names() for vectors or dimnames() for matrices

Data Types – Summary

- Summary of the Data Types videos, no additional information

Reading Tabular Data

- The most common read data functions are as follows:
    - read.table() and read.csv() for reading tabular data, returns a dataframe in R
    - readLines() for reading lines of a text file, returns a character vector in R

    - source() for reading R code files
    - dget() for reading deparsed R code files
    - load() for reading in saved workspaces
    - unserialize() for reading single R objects in binary form
- There are analagous functions for writing data in R (write.table(), write.csv(), dump(), dput(), save(0 and serialize())
- The arguments of read.table are as follows:
    - File – the name of a file or connection
    - Header – logical indicating if the files has a header line
    - Sep – string indicating how the columns are separated
    - ColClasses – character vector indicating the class of each column in the dataset
    - Nrows – number of rows in the dataset
    - Comment.char - character string indicating the comment character
    - Skip – the number of lines to skip from the beginning
    - StringsAsFactors – logical for if the strings should be treated as factors
- read.csv is identical to read.table but the default separator is a comma
- Setting the colClasses can massively reduce runtime
- You can quickly find the classes via:

```
x <- read.table("data", nrows = 100)
classes <- sapply(x, class)
Data <- read.table("data"), colClasses = classes)
# This loads the first 100 rows, loops over the column to detect their classes and passes this to read.table
```

- To estimate the size (in number of bytes) of a dataset do columns x rows x8 and divide by 2^{30} to convert to GB
- You need roughly twice as much RAM as the size of dataset to read in a dataset to R

<u>Reading Large Tables</u>

- read.table() can struggle to read large datasets, but there are steps that can be taken to reduce the time R takes to read in data
- Ensure the dataset is not larger than the RAM on your PC
- Set comment.char = "" if there are no comments
- See the help page for read.table for many hints

<u>Textual Data Formats</u>

- There are other text-based formats beyond tabular and csv
- dumping() and dputing() are used to read this data and are useful since the resulting format is editable and potentially recoverable
- These functions preserve metadata
- Textual formats generally work better with version control programmes such as git which can only track changes meaningfully in text files; hence it can be easier to fix issues

- However, the format is not very space efficient so often need to be compressed
- dput() takes most R opjects and constructs R code which allows you to read in the data
- Usually, you want to save this code to a file and then use dget to read in the data
- dump is similar to dget but can be passed multiple objects rather than just one
- You can use dump to write data to a file and then if it ever gets removed you can bring it back with source()

Connections: Interfaces to the Outside World

- Connections can be made to individual files using file(), gzip files using gzfile(), bzip2 files using bzfile() and webpages using url()
- Check out the help file for file() to see the arguments
- Connections can allow you to navigate files or external objects in sophisticated way though in practice we often don't need to deal with the connection directly
- One example where it can be useful is when you want to just read part of a file (e.g. setting up a connection to a webpage and reading the first 10 lines of HTML)

Subsetting - Basics

- The single square bracket [] always returns an object of the same class as the original and can be used to extract multiple elements (e.g. subsetting the vector x <- c(1:3) via x[2] returns a vector of length 1 with value 2)
- The double bracket [[]] is used to extract elements of a list or dataframe (which are not necessarily another list or dataframe) and can only extract a single element
- The dollar $ is used to extract named elements of a list or dataframe
- You can use comparisons to extract elements which meet a specific criteria (e.g. x <- c(1:5) x[x > 3])
- What this is effectively doing is creating a logical vector x > 3 and using this to extract elements of the vector x; hence you can use any logical vector of the same length to subset a vector

Subsetting - Lists

- Lists can be subsetted with the single or double square brackets, the single square bracket will return another lists while the double square bracket will return the object in its true class
- You can use the double square brackets with an objects name to extract it from a list
- Using the $ to subset named elements is useful since you don't have to remember it's place in a list, this is especially useful for large lists and dataframes

- If the index you desire is the result of some kind of computation then you must use the double brackets not the $
- If you have a lists nested within a list then you can subset them by passing a vector with the double brackets
  e.g. x <- list(a = list(1, 2, 3), b = c(10, 11, 12))
  x[[c(1, 3)]] = 3 or x[[c(2, 3)]] = 12

Subsetting - Matrices

- Matrices are subseted using single square brackets with two arguments corresponding to the to the row and column index
- If either index is missing the entire row/column will be returned
- When a single element is retrieved from a matrix, a vector of length 1 is returned rather than a 1x1 matrix; this is the 1 exception to the rule that the single bracket always returns an object of the same class; setting drop = FALSE will prevent this and return a 1x1 matrix
- The same applies for extracting a single row or column

Subsetting – Partial Matching

- By default the $ operator looks for match though it can sometimes find a match even if the entire name is not provided
  e.g. x <- list(abcdefg = c(1:5), hijklmn = matrix(1:4, 2, 2))
  x$a = 1 2 3 4 5
- However by default the double bracket operator looks for an exact match; setting exact = FALSE removes this though be careful to ensure you are extracting what you actually want

Subsetting – Removing Missing Values

- Start by creating a logical vector which tells you which values are NA via is.na()
- Pass this to the single bracket operator [] with a ! Before to remove those values
  e.g. x <- c(1, NA, 3)
  bad <- is.na(x)
  x[!bad] = 1 3
- complete.cases() is used when you have multiple objects with missing values in different places and you want to only take the rows which have full data for each place

Vectorised Operations

- Vectors of the same length and matrices of the same dimensions can be added, subtracted, multiplied and divided and the operations will be evaluated element-wise
- Comparative operators such as < or == will return logical vectors where the comparison is evaluated element wise

- If you want true matrix multiplication you must use %*%

Introduction to Swirl

- Swirl is an interactive package built by a student at the Johns Hopkins University that allows users to learn R coding and statistics within R
- User can answer questions by typing lines of code or selecting answers from multiple choice
- Typing skip() allows you to skip the current question.
- Typing play() lets you experiment with R on your own; swirl will ignore what you do until you type nxt() which will regain swirl's attention.
- Typing bye() causes swirl to exit. Your progress will be saved.
- Typing main() returns you to swirl's main menu.
- Typing info() displays these options again.

Swirl Lesson 1 – Basic Building Blocks

- Arithmetic operations between vectors are evaluated elementwise and when the vectors are of different lengths the smaller vector will be cycled through until all operations are complete
- If the length of the smaller vector does not divide the length of the longer vector then R will produce a warning (but will still evaluate the command)

Swirl Lesson 2 – Workspace and Files

- getwd() shows the current working directory
- ls() lists all objects in your local workspace
- list.files() or dir() list all files in your working directory
- args() lists the arguments of a supplied function
- dir.create() create a new directory in the current working directory
- setwd() changes the working directory to a supplied location
- file.create() to make a new file in the current directory
- file.exists() tests if a file can be found within the working directory
- file.info() provides macros data about a file; the result is a list so you can use $ to extract specific macro data
- file.rename() to change the name of a file
- file.copy() to copy file
- file.path() lists the location of a file (independent of operating system)

Swirl Lesson 3 – Sequences of Numbers

- To search the help files for an operator such as : you must enclose it in ` via for example ?`:`
- seq() to create sequences of numbers
- seq_along create a sequence of length the same as the length of its argument

- rep() to create a sequence with the same value (note the value can itself be a vector)
- Using the each argument in rep() allows you to produce a vector which has all the repetitions of the first value then all the repetitions of the second value and so on

Swirl Lesson 4 – Vectors

- != is the logical for not equal
- A | B is used for “A or B” where A and B are logicals
- Similarly A & B is used for “A and B”
- !A is used for “Not A”
- paste() to join character vectors

Swirl Lesson 5 – Missing Values

- Using the equals operator == with NA values will always return NA
- R encode TRUE as 1 and FALSE as 0 so can combine sum() and is.na() to count missing values

Swirl Lesson 6 – Subsetting Vectors

- Index a vector using [ ]
- Use is.na to index missing values and combine with ! to get the actual values
- R uses ‘one-based indexing’ meaning the first element of a vector is considered element 1 (unlike other languages where it is considered the 0 element)
- Indexing using negative integers will cause R to include all element except the ones indexed
- Name each element of a vector wiih names()
- Provide names within [ ] to subset by name

Swirl Lesson 7 – Matrices and Data Frames

- Apply dimension to a vector via dim(); this will convert it to a matrix
- class() to find the type of an R object
- matrix() to create a matrix noting elements are entering column-wise
- cbind() to combine column-wise (note this may coerce values to all be of the same class)
- data.frame() to create a data frame which can contain different classes of data
- colnames() to assign column names to a dataframe (works similarly to dim())

Control Structures - Introduction

- Control structures allow you to govern the flow of execution of a programme
- If, else: Test 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: stop the execution of a loop

- Next: skip an iteration of a loop
- Return: exit a function

Control Structures – If-Else

- The sytax in R is as follows:
  If(<logical condition 1>) {
  ## do something
  } else if(<logical condition 2) {
  ## do something different
  } else {
  ## do seomthing else
  }
- You can assign the value of the output of an if-else statement inside the statement or assign the whole statement to a variable

Control Structures – For Loops

- The syntax in R is as follows
  For(i in sequence) {
  # do something for this iteration
  }
- Note, the { } may be ommit if there is only a single expression in the loop
- R can index for loops either by index number, using seq_along, letter (for character vectors)
- For loops can be nested but be careful since this can make code hard to understand and can lead to slow run times

Control Structures – While Loops

- Syntax in R is:

While(logical expression) {
# do something which potentially could change the logical condition
}

- Be careful that the logical condition will eventually return FASE otherwise the loop will run forever

Control Structures – Repeat, Next, Break

- Repeat initiates an infinite loop which can only be ended with break
- Syntax in R is

Repeat {
# do something
if(logical condition involving above calculation){
break
} else }
# do something different
}
}

- Often used for estimation functions where you want to estimate something within a desired degree of error
- Be careful that the loop will actually end
- next is used to skip iterations in a loop
- return signals that a function should exit and return a given value

Your First R Function

- The syntax for functions in R is:

```
functionName <- function(arguments){
# do something with those arguments
}
```

- Functions will return the last expression by default but you can use return to output other values if needed
- Specify default values for arguments by assigning them a value in the creation of the function

Functions (part 1)

- Functions in R have the class “function” and are treated like any other R object; in particular, they can be passed as arguments to other functions and can be nested (i.e. defined within another function)
- Formal arguments are the arguments oncluded in a function definition (formals() returns these)
- Function arguments can be missing or might have default values
- Arguments are matched positionally or by name when calling a function (in practice, it’s best not to mess with the order of arguments)
- Function arguments can be partially matched by name (the order of matching goes exact, partial, positional)

Functions (part 2)

- You can set an argument value to default to NULL
- R makes sue of ‘lazy evaluation’ where arguments are only called upon when needed so unused arguments will not prevent a function from running
- The argument “...” indicates a variable number of arguments and is usually used to pass arguments onto another function
- For example, if you wanted to create a function which modified some of the default argument values of a function with many arguments, you would not want to type out all the arguments of the original function so you use “...”
- “...” can also be used when the number of arguments cannot be known in advance e.g. paste()
- Any arguments after “...” must be explicitly named since how would R know it was not another argument meant for “...”

Scoping Rules – Symbol Binding

- If a symbol has conflicting values R searches through a series of environments to find the value; first the global environment and then any

packages in the search list ending on the base package (use search() to list the environments)
- Environments can be thought of as lists of symbols and their values
- Loading a package with library() put that package on position 2 in the search list by default
- R has separate namespaces for functions and non-functions so it's possible to have a function and variable with the same name
- Free variables are variables which appear in a function but are not arguments
- R uses lexical scoping which is where the value of free variables is searches for in the environment where the function was defined; if it's not found there then the search is continued in the parent environment and so on

Scoping Rules – R Scoping Rules

- In R you can have function defined inside other functions (unlike languages such as C); in this case the environment in which a function is defined is the body of another function
- Use ls() to see what's in a function's environment
- Other languages used dynamic scoping where the value of a free variables is searched for in the environment from which a function was called
- As a result of lexical scoping, all R objects must be stored in physical memory and all functions must carry a pointer to its defining environment which could be anywhere

Scoping Rules – Optimisation Example

- Many optimisation routines in R require you to pass a function whose first argument is a vector of parameters over which optimisation is to take place
- However, the function may depend on things besides the parameters such as data; also, it may be desirable to fix certain parameters
- Most optimisation function in R attempt to minimise by default
- See example in video
- Effectively, objective functions can be built which contain all the necessary data for evaluating the function without the need for long argument lists

Coding Standards

- Write code using a text editor and save it as a text file as this is a 'lowest common denominator' across different operating systems and programmes
- Indent your code (using crtl i) to make it more readable; typically indent functions, if-else statement and loops
- Limit the width of your code (~80 columns)
- Limit the length of individual functions to one basic task; this makes debugging easier as errors can be traced back to one specific function

Dates and Times

- Dates are represented by the date class and are stored internally as the number of days since 1st January 1970

- Times are represented by the POSIXct or POSIXlt class and are stored internally as the number of seconds since 1st January 1970
- Use as.date to coerce dates to the date class
- weekdays(), months() and quarters() are useful basic functions which convert dates to day of the week, month of the year, quarter of the year respectively
- Sys.time() gives current time
- As.POSIXlt to convert to POSIXlt which stores time as a list with named columns
- strptime() converts the format of dates (e.g. swapping order of month and day)
- The usual operators (+, - , ==, etc.) all work with dates and times
- R even tracks leap years, leap seconds, daylight savings and time zones

Swirl Lesson 8 – Logic

- == is used to test if two objects are equal and != to test if two objects are not equal
- & is used for AND and | is used for OR
- & or | evaluates across a vector whereas && rr || only evaluates the first element
- The equivalent of BODMAS for logical operators is that all AND expressions are evaluates before OR expressions
- isTRUE() and isFALSE() evaluate a logical statement
- identical() returns TRUE if its two arguments are identical
- xor() is the exclusive or function and evaluates to true if and on if exactly one argument is TRUE
- which() takes a logical vector and returns the true indices
- any() takes a logical vector and returns TRUE if any of the values are TRUE
- all() takes a logical vector and returns TRUE if and only if all of the values are TRUE

Swirl Lesson 9 – Functions

- It's best practice to specify arguments in order even though R can sometimes deal with the order permuted to avoid error and make code readable
- args() show the arguments of a function
- You can pass a function as an argument without first defining the function, these are called anonymous functions
- A%%B returns the remainder of A/B
- You can define a new binary operator using "%name%" and define it as you would a function; you can then call arg 1 %name% arg 2

Swirl Lesson 10 – lapply and sapply

- These functions follow the split-apply-combine strategy
- lapply() stands for list apply; it takes a list as input, applies a user defined function to each element of the list then returns a list of the same length as the original one

- sapply() stands for simplify apply; it implements lapply and then simplifies the output into a vector rather than a list
- If the result of sapply is a list where every element is of length one, then it returns a vector; if the result is a list where every element is a vector of the same length (> 1), sapply() returns a matrix.
- You can define the function to apply within sapply() or lapply(); functions defined in this way are not saved to the environment, they effectively dissapear when the function has evaluated

Loop Functions - lapply

- Loop functions are used to execute a loop in a compact form (i.e. with less lines of code)
- lapply: Loop over a list and evaluate a function on each element
- sapply: Same as lapply that tries to simplify the result (it will return a list if it cannot simplify in a 'sensible' way)
- apply: Apply a function over the margins of an array
- tapply: Apply a function over subsets of a vector
- mapply: Multivariate version of lapply
- You can define the function to apply within the function call
- If the result of sapply is a list where every element is length 1, then a vector is returned.
- If the result of sapply is a list where every element is a vector of equal length (>1) then a matrix is returned

Loop Functions - apply

- apply is used to evaluate a function over the margins of an array (often to the rows/columns of a matrix which is a 2-dimensional array)
- Not necessarily faster than a loop but is more succinct in terms of the amount of code
- Some simple applications of apply have shortcuts (e.g. row/colSums/Means) which are highly optimised for large matrices
- The MARGIN argument is apply is which dimensions you would like to preserve

Loop Functions - mapply

- mapply is a multivariate version of lapply/sapply used to apply a function in parallel over a set of arguments
- Whereas sapply/lapply can only apply a function over a single object, mapply can evaluate a function over multiple objects

Loop Functions - tapply

- tapply is used to apply a function over subsets of a vector
- The argument INDEX is a factor of the same length as the vector to which the function will be applied which identifies which group each element of the vector belongs to

- gl() can be used to generate factor variables

Loop Functions - split

- split is not a loop function but is useful in conjunction with loop functions
- It segments out a vector or list (including a data frame) based on the factor argument which is passed to the function
- It always returns a list
- Often split is sued to segment out data and then lapply to apply a function to each group
- split is often used with data frames where column can be called by name (e.g. calculating the mean temperature in a particular month)
- interaction() can be used to create every combination of multiple factor variables (note this can create empty levels which can be ignored with the DROP = TRUE argument)

Debugging Tools – Diagnosing the Problem

- You can tell there may be a problem by
    - Message: A generic notification that does not stop the execution of the function
    - Warning: An indication that something is wrong but that it did not stop the function from running
    - Error: A fatal problem has occurred which prevented the function from running
    - Condition: A generic concept for indicating that something unexpected could occur (programmers can create their own conditions)
- To know if there is something wrong with a function ask:
    - What was your input? How did you call the function?
    - What were you expecting?
    - What did you get?
    - How do the above two differ?
    - Were your expectations correct in the first place?
    - Can you reproduce the problem?

Debugging Tools – Basic Tools

- The tools in R for debugging are:
    - Traceback: Prints out the function call stack after an error occurs so you can see exactly where the error occurred within the call
    - Debug: Allows you to step through the execution of a function one line at a time
    - Browser: Suspends the execution of a function whenever it is called and puts the function in debug mode
    - Trace: Allows you to insert debugging code into a function in specific places
    - Recover: Allows you to modify the error behaviour so that you can browse the function call stack

- You can also put print statements in your function so that you can see the value of different variables within the function call

Debugging Tools – Using the Tools

- Debug won't say what the error is, but it will allow you to find the line within the function call where the error occurred so you can narrow the source of the issue down
- options(error = recover) will set the global error option to recover; now whenever an error occurs you will get back the function call stack (i.e. what you would get if you called traceback after the error) rather than the usual console
- Debugging tools are not a substitute for thinking

Swirl Lesson 11 – vapply and tapply

- Whereas sapply() tries to 'guess' the correct format of the result, vapply() allows you to specify it explicitly (an error will be produced if the result doesn't match the format you specify)
- tapply splits the dataset up into groups and then applies a function to each group
- table() returns the number of each data points in each level of a factor

The str Function

- str() compactly displays the internal structure of an R object; especially suited to nested lists
- A useful alternative to summary(), roughly displaying one line per basic object
- Can be used when first loading a dataframe to understand what the dataset consists of
- table() returns the number of each data points in each level of a factor

Simulation – Generating Random Numbers

- rnorm: Generate random normal variates with a given mean and standard deviation
- dnorm: Evaluate the normal probability density with given mean and standard deviation at a point or vector of points
- pnorm: Evaluate the cumulative distribution function for a normal distribution
- qnorm: The inverse of the pnorm function
- There are analogous function for other probability distributions (e.g. rpois, dpois, ppos and qpois for the Poisson distribution)
- Set.seed: Sets the random number seeds to allow you to reproduce the results of random number generation

Simulation – Simulating a Linear Model

- Errors in a linear model are generally assumed to be normally distributed except for say when the outcome Y has a Poisson distribution in which case log of the rate variable μ follows a linear model

- See examples from video

Simulation – Random Sampling

- sample: Allows you to draw randomly from a specified set of objects
- The argument replace sets whether you allow for repeated samples or not (i.e. whether objects are replaced before resampling or not)
- Remember to set the seed when doing random sampling

Simulation – R Profiler (part 1)

- A tool for helping to optimise R code which is taking a long time to run
- Profiling is a method for examining how much time is spent running different parts of a programme and hence is useful when trying to optimising your code
- In order to get the biggest impact on speeding up your code, you need to first know where the code spends most of its time; this cannot be done without profiling
- Focus on getting code to run properly first and then optimise rather than trying to optimise as you go
- system.time returns the amount of time in seconds to evaluate a given expression (or the time until the first error occurs)
- User time is the amount of time charged to the CPUs whereas elapsed time is the true 'wall-clock' time
- Often user time and elapsed time are very similar for simple tasks
- Elapsed time may be great than user time if the CPU spends time waiting around (e.g. for processes outside the programme to finish)
- Elapsed time may be small than user time if the computer can access multiple processors
- Elapsed Time > User Time when reading data from the web
- Elapsed Time < User Time when using parallel computing or libraries optimised for computational expensive processes
- system.time assumes you know where to look for bottlenecks which may not always be the case, particularly for very large programmes

Simulation – R Profiler (part 2)

- Rprof starts the profiler for R and summaryRprof summarises the output from Rprof
- Rprof keeps track of the function call stack and records how long each function took
- There are two methods for normalising the data for summaryRprof:
    - "by.total": Divide the time spent in each function by the total run time
    - "by.self": Does the same as by.total but first subtracts out the time spent in any functions above in the call stack
- "by.total" is often not as useful as "by.self" because many functions do most of the work in lower-level functions which are called by the top-level function. Hence, it makes sense to first subtract out the time spent by these lower-level functions to understand how much real work the top-level function is doing

Swirl Lesson 12 – Looking at Data

- Class, dim, nrow and ncol are all useful for looking at the size and form of a dataset
- object.size tell you how much storage space an object takes
- Names: returns a character vector of variable names
- summary and str are useful summarisation functions for datasets