A Primer on R for Data Munging and EDA

Tyrone Ryba

August 26, 2021

Contents

1 R primer part I **2**
- 1.1 Introduction to R language features . . . 2
- 1.2 Sources of help . . . 3
- 1.3 R books . . . 4
- 1.4 Data structures . . . 4
  - 1.4.1 Vectors . . . 5
  - 1.4.2 Matrices . . . 8
  - 1.4.3 Data frames . . . 8
  - 1.4.4 Lists . . . 9
- 1.5 Data types . . . 9
  - 1.5.1 Logical . . . 10
  - 1.5.2 Integer . . . 10
  - 1.5.3 Double . . . 11
  - 1.5.4 Character . . . 11
  - 1.5.5 Factor . . . 11
- 1.6 Examining R Objects . . . 12

2 R primer part II **13**
- 2.1 Program flow . . . 13
  - 2.1.1 Conditional statements . . . 13
  - 2.1.2 For loops . . . 15
  - 2.1.3 While loops . . . 16
  - 2.1.4 Syntax for flow control and conditional statements . . . 16
- 2.2 Functions . . . 17
  - 2.2.1 Getting code for a function . . . 19
  - 2.2.2 Modifying a function . . . 19
- 2.3 R working directory . . . 19
- 2.4 R working environments and memory . . . 20
  - 2.4.1 Environment namespaces . . . 20
  - 2.4.2 Environment variables . . . 20
  - 2.4.3 Garbage collection . . . 21

1 R primer part I

R is a language designed for interactive data analysis that developed from the S language from Bell Laboratories. R's strengths are in its expansive ecosystem of user-created packages that extend the core language, available through the Comprehensive R Archive Network (CRAN), Bioconductor, and GitHub, and used in a broad collection of workflows in data science and other fields.

This section will cover the basics of installing and administering R, as well as methods for getting help and common R data structures.

1.1 Introduction to R language features

As a language, R was written with statistical programming in mind, and as a result has several design quirks that are unexpected for users of more traditional programming languages like C++, Python, or Java.

R scripting – R is primarily a scripting language, and interpreted line-by-line. This can be quite useful; scripts can be assembled into more substantial programs or compilations of analysis methods, but debugged and extended interactively at each line.

R packages – Much of the R code useful for projects was not written by the R Core Team (who designed base R), but by the community. There is now an enormous variety of R packages available, and the ease with which packages can be developed and released has played a major role in R's popularity. Lists of packages organized by topic are available through CRAN and Bioconductor, most recent packages are also hosted on general code repositories like GitHub.

R packages can be thought of as libraries for other languages, as they contain related functions designed to work together. One difference is that packages are often designed to streamline a common workflow. For example, the `affy` and `limma` packages are popular for preprocessing, normalizing, and collecting results from certain gene expression datasets.

R workflows – R's particular strength is in interactive data analysis. A related strength, with packages like knitr and R markdown, is in making the process and results of interactive analysis easy to reproduce and report to others. One effective way to keep track of our topics in class would be to save a copy of our .Rmd guides with notes alongside related code sections in an R Markdown document (`File` ⇒ `New File` ⇒ `R Markdown`). Commands to customize layout in RStudio can be found under `Tools` ⇒ `Global Options`.

Installing R

Linux: Most Linux distributions come with a version of R preinstalled. An R session can be started from the command line with "`R`". If not yet present, R can generally be installed with "`sudo apt-get install r-base`" for apt, "`pacman -S r-base`" for pacman, etc. As for other packages, you will often want to first check that the local database for your package manager and system are up-to-date.

Mac/Windows: For Mac or Windows machines, new precompiled binaries can be found for stable or development versions of R at CRAN.

R Development Environments: Rather than the default R terminal or GUI, most prefer to work in a specialized environment, like a programmer's text editor or R-centric integrated development environment (IDE).

As of 2021, RStudio remains the most popular IDE by a good margin, and balances usability and a wide variety of language-specific features. There are many alternatives, including Microsoft R Open (once Revolution R), which is built on a Visual Studio framework but with emphasis on applications in business intelligence and distributed computing. There are also popular R plugins for general purpose IDEs and text editors, including Eclipse, SublimeText, Emacs, and Vim.

1.2 Sources of help

Quick-R – Lots of helpful examples that are categorized and easy to copy into R. A good place to start.

The R Wikibook – A fairly comprehensive wiki with contributed examples of practical problems.

R Mailing Lists – People often post here for help on specific problems, and other users and some R developers contribute helpful suggestions.

Rtips – Example code for a long list of common R questions.

The R Reference Card, by Tom Short – This is a nicely categorized list of the most often used R commands with brief descriptions.

StackExchange (and similar sites) – A great Q/A site in general, and with a fairly large selection of common question on R.

- StackOverflow: Programming-related questions
- BioStars: Biology/bioinformatics questions
- CrossValidated: Statistical questions

R abbreviations explained – At first (and even after a while), the names of functions in R can seem a bit obscure if you don't have a Linux/Unix background. This post has a nice guide to expanded abbreviations that will help you understand the etymology.

R Manuals – R-project.org has an extensive collection of guides from the authors of the core language. These are nicely organized, but more reference-style and in-depth than you may need at first.

R itself – Just enter "`?`" in front of a function name to bring up its corresponding help page, e.g., "`?data.frame`". All help pages have examples at the bottom that you can copy into the R console to get a feel for the commands. You can also use `help.search()` or `apropos()` for a general topic or related function.

1.3 R books

Bookdown

There are many recent books written in bookdown that will be useful for your current and future work in data science. As of Fall 2021, these include:

1. *R for Data Science*, Grolemund and Wickham
2. *Advanced R*, Hadley Wickham
3. *Text Mining with R: A Tidy Approach*, Silge and Robinson
4. *Hands-On Programming with R*, Grolemund
5. *R Programming for Data Science*, Roger Peng
6. *Advanced Statistical Computing*, Roger Peng
7. *Practical Data Science with R*, Zumel and Mount
8. *R Cookbook*, Long and Teetor
9. *Data Science at the Command Line*, Jeroen Janssens
10. *Bookdown: Authoring Books and Technical Documents with R Markdown*, Yihui Xie

Many other specialized texts can be found under the "Books" tab at https://bookdown.org/.

1.4 Data structures

Like other languages, data in R are stored in variables, and each variable is a container for data in a particular structure. There are four data structures most commonly used in R: Vectors, matrices, data frames, and lists.

For many functions, this data structure determines which of several alternative paths are used. The easiest way to query the structure of an R variable is using `str()`. The `str()` function will return the type of structure (data frame, list, vector, matrix, etc.), and its dimensions, including number of columns (variables) and number of rows (observations). Along with the Environment tab in R IDEs like RStudio, `str()` is a convenient way to keep track of changes as variables are processed.

```
  # Define a new variable, x
  # Check the structure of the variable
  x = 1
  str(x)
```

```
##  num 1
```

Here, x is a numeric data type, "`num`", with one value, "`1`".

1.4.1 Vectors

The most basic data structure in R is the vector, which is built from an analogous container structure in C. Vectors in R are entirely unrelated to mathematical vectors, and map more closely to sets. Vectors have a length that can be queried with length(), and a single data type (an atomic class) for every element.

```
  # Define a vector of integers
  vec <- c(1,5,7,9,4,8,3)
  vec
```

```
## [1] 1 5 7 9 4 8 3
```

```
  str(vec)
```

```
##  num [1:7] 1 5 7 9 4 8 3
```

One behavior to be careful of is R's implicit type conversion. If any element in a matrix or vector is changed to a different type, every element is converted to the type that could store all elements as that type. This follows a general path from logical values to numeric values to character values.

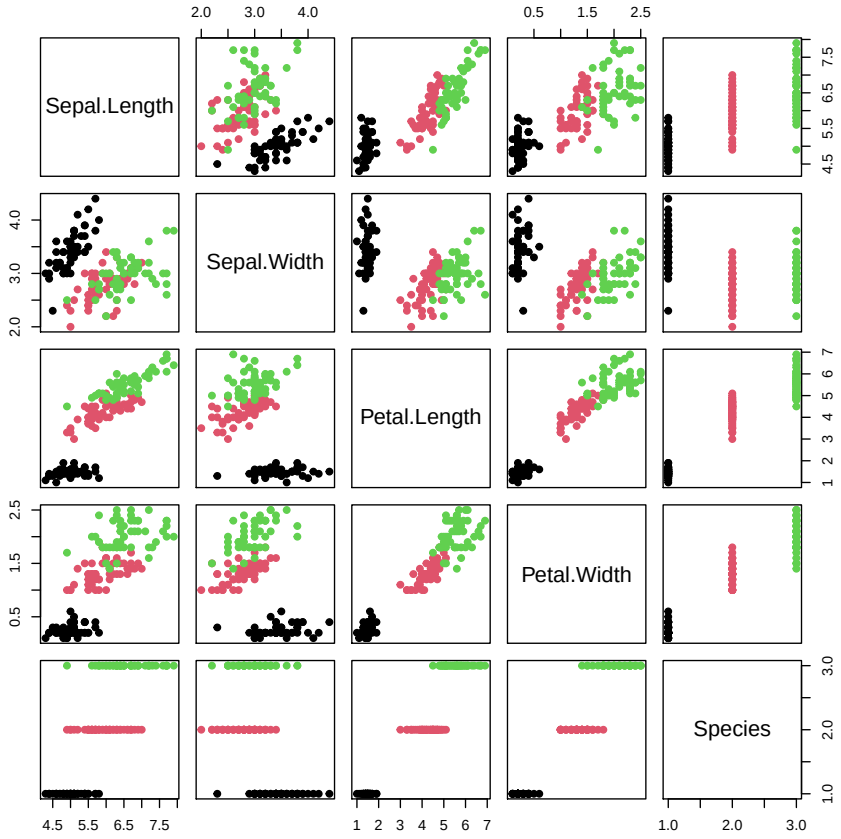
```
  # Replace one number with a character value
  vec[3] = "wrench"
  str(vec)
```

```
##  chr [1:7] "1" "5" "wrench" "9" "4" "8" ...
```

This automatic conversion, while often unexpected at first, can also be useful. For example, we can create summary statistics or plots from logical values without an explicit conversion.

Importantly, R is a language built on *vectorization*. By default, most R functions will work on vectors, and even larger data containers (matrices and data frames) just as easily as they would on a single value. In fact, their behavior for single values is often defined by that for a vector with one element. Here is an example of a single plot command that produces a matrix of plots, and another function to create a matrix of correlations:

```
# Plot the iris dataset
# col: Color by species
# pch: Use filled points
plot(iris, col=iris$Species, pch=19)
```



```
  # Return the Pearson correlations
  # among iris dataset columns 1-4
  cor(iris[,1:4])
```

```
##              Sepal.Length Sepal.Width Petal.Length
## Sepal.Length    1.0000000  -0.1175698    0.8717538
## Sepal.Width    -0.1175698   1.0000000   -0.4284401
## Petal.Length    0.8717538  -0.4284401    1.0000000
## Petal.Width     0.8179411  -0.3661259    0.9628654
##              Petal.Width
## Sepal.Length   0.8179411
## Sepal.Width   -0.3661259
## Petal.Length   0.9628654
## Petal.Width    1.0000000
```

Here, we read a dataset that was already defined in R, though not yet loaded in main memory. R uses a system of deferred, or lazy, evaluation. We will talk more about this when we get to optimization.

There are several unusual features of vectors in R. First, vector indices start at 1 rather than 0, which is the standard in most other languages.

```
  vec <- c(1,5,7,9,4,8,3)
  vec[2]    # Second element of the vector
```

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

Second, vector indices can go beyond the length of the vector. This would normally give an "Index out of range" error in other languages, but R happily inserts NAs (missing values) where indices exceed the vector length. This is one area where defensive programming practices are useful: we should check row and column lengths before indexing unknown vectors.

```
  vec
```

```
## [1] 1 5 7 9 4 8 3
```

```
  vec[7:10] # NAs produced when index is out of range
```

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

Third, negative indices do not refer to elements in reverse order, but remove those elements from the vector. This is one way to indicate we want "everything but x", or another set within the vector.

```
  vec
```

```
## [1] 1 5 7 9 4 8 3
```

```
  vec[-2]   # Not 8 or NA! Negative signs remove indices
```

```
## [1] 1 7 9 4 8 3
```

```
  vec[-c(1,3,5)] # Remove elements 1,3,5
```

```
## [1] 5 9 8 3
```

Fourth, vector indices of boolean values (TRUE and FALSE) can wrap. Wrapping of vector indices can be dangerous if unintentional, but is often useful. For example, we can use the following to extract every third element of a vector with a length that is a multiple of three.

```
  # Collect every third value
  vec[c(FALSE, FALSE, TRUE)]
```

```
## [1] 7 8
```

1.4.2 Matrices

Matrices behave as two-dimensional vectors. They are faster to access and index than data frames (described below), but less flexible in that every value in a matrix must be of the same atomic data type (integer, numeric, character, etc.). If one is changed, implicit conversion to the next data type that could contain all values is performed.

```
  mat = matrix(1:100, nrow=10, ncol=10)

  rowMeans(mat)
```

```
##  [1] 46 47 48 49 50 51 52 53 54 55
```

```
  colMeans(mat)
```

```
##  [1]  5.5 15.5 25.5 35.5 45.5 55.5 65.5 75.5 85.5 95.5
```

Matrices are indexed by row, then column. Indexes can include numeric sequences or TRUE/FALSE values.

```
  mat[1:3, 1:3]
```

```
##      [,1] [,2] [,3]
## [1,]    1   11   21
## [2,]    2   12   22
## [3,]    3   13   23
```

1.4.3 Data frames

One of the main features of R upon its introduction was its ability to work with matrices in which each column was a different data type – the data frame.

Internally, data frames operate as a special type of list, as described in the next section, but the tabular nature of data frames (versus the unstructured nature of lists) allows for a number of functions to apply to them.

```
  head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

```
  iris$Species[1:10]
```

```
##  [1] setosa setosa setosa setosa setosa setosa setosa setosa
##  [9] setosa setosa
## Levels: setosa versicolor virginica
```

1.4.4 Lists

Lists are simply vectors of other objects. This means that we may have lists of matrices, lists of dataframes, lists of lists, or a mixture of essentially any data type.

This flexibility makes lists a popular structure for storing objects in most R packages. Custom classes for package objects are often built from stacks of matrices and data frames with expected dimensions and types stored in lists. From our previous example, we might create a list that combines the N x M matrix of data values with the N x O matrix of row annotations and M x P list of column annotations.

1.5 Data types

Data structures are composed of elements that are *atomic data types*. There are five of these that are common in R:

1. Logical
2. Integer
3. Numeric
4. Character
5. Factor

1.5.1 Logical

Logical values (or Boolean) describe logical states that are exclusively true or false. In R, they are very commonly used to index variables and match cases that meet particular combinations of conditions. Examples will be covered in more detail later, but below are two common applications of logical indices and value tests.

```
  a <- c(TRUE, TRUE, FALSE)
  b <- c(FALSE, TRUE, TRUE)
  c <- c(FALSE, TRUE, FALSE)

  a & b
```

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

```
  a | c
```

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

```
  xor(a, c)
```

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

Notice that these comparisons are vectorized: with one logical operation, we compare every matching element of both (or many) variables.

The characters "T" and "F" can be substituted as shorthand for "TRUE" and "FALSE", but this can invite errors. See what happens when the values for T and F are reassigned.

```
  if(T) cat("Condition true. \n")
```

```
## Condition true.
```

```
  T = "tomato"
  F = "Fahrvergnügen"

  # if(T) cat("Condition true. \n")
```

1.5.2 Integer

Integer values are used for storing positive or negative whole numbers. You can specify that data will be stored as integers by using the as.integer() function, or by the shorthand of appending "L" to a number.

```
  x <- as.integer(300)
  x <- 300L
```

Some errors can arise from the distinction between integer and double precision numeric values. In some cases, whole numbers that appear nicely suited to storage as integers will require more space than integers can accommodate. Try storing an integer of the current world population estimate (7.9 billion) in an R variable.

Standard integers are stored with 32-bit precision in R, so these can only represent numbers up to roughly 2 billion (2^{31}, with one bit used for sign). As a workaround for vectors with more than two billion entries, R recently switched to using numeric values as indices for vectors. This can have a few interesting side effects to be aware of, as illustrated below.

```
  c(1, 5, 7)[2.6]
```

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

1.5.3 Double

Numeric ('num') values are generally doubles, or double-precision floating point vectors. Compared to integer values, floating point values require roughly twice the memory space. In previous versions of R and S, the number of digits stored could be machine-specific. Current R versions require conformity to the IEC 6055 standard, corresponding to at least 2^{308}, but with differences possible per machine depending on the details visible in .Machine. You can check these details for your own machine by entering '.Machine' in the R console.

1.5.4 Character

Character values in R are strings of one to many alphanumeric characters or symbols. Aside from storing categorical text values or strings, characters ('chr') can be useful for concatenating output to the terminal for status messages or debugging using cat() or paste(). Substrings or substitutions are possible with substr() and sub() or gsub, respectively, and strings can be searched with grep().

1.5.5 Factor

Factors often begin as character vectors, and are much like dictionaries in Python and similar to hash tables (key-value stores) more generally. A vector stored as a factor will have each unique value mapped to an index. This index can then be stored in the place of individual data values, which are often more difficult to store or sort.

In several situations R will automatically convert character vectors to factors. For example, the default for reading regularly formatted tabular text, read.ta-

ble(), is the parameter setting stringsAsFactors=TRUE. All of the object types described so far affect which functions will apply and how they will behave, so it will be important to know how an object or variable is stored for later processing steps. Fortunately there are many ways to examine object types and structures in R.

1.6 Examining R Objects

There are several useful functions to examine the details of R objects.

1. str(): Examine the structure of an object.
2. length(): Return the number of elements in a vector-like object.
3. dim(): Return the dimensions of a matrix-like object.
4. class(): Return the high-level class of an object ("data.frame", "matrix", etc.)
5. typeof(): Return the lower-level type of an object ("list", "double", etc.)
6. summary(): Return a summary of an object's elements that depends on its class. See summary(iris).

In addition to the family of conversion functions (as.character, as.integer, as.factor, as.numeric, etc.), there are corresponding type checking functions (is.character, is.logical, is.infinite, is.na, is.null, etc.) that can be useful within blocks of code that anticipate particular data types. All of these are vectorized, so we can easily check and convert all values stored in a variable at once.

2 R primer part II

Today we continue our overview of R programming, and will cover the ways in which the execution of R code can be repeated or made conditional. We will also go over some more general aspects of the R environment, workspaces, and namespaces.

2.1 Program flow

2.1.1 Conditional statements

Conditional statements change the program flow or the contents of a variable based on a specified condition. For example:

```
If a condition is true, run a block of code
  else, run a different block of code
If a condition is true, assign a variable a certain value
  else, assign a different value
```

Recall that most functions in R are *vectorized*, and operate on vectors of values similarly to individual values. In contrast, conditional statements operate on only the first condition in a vector. The following returns a warning:

```
cond = 0
if(c(TRUE, FALSE, TRUE, TRUE, FALSE)) {
  cond = cond + 1
  cat("Condition ", cond, "is true.\n")
}
```

```
## Warning in if (c(TRUE, FALSE, TRUE, TRUE, FALSE)) {: the condition
has length > 1 and only the first element will be used
```

```
## Condition  1 is true.
```

For output to the console (the default in R), an alternative is to combine vectorized statements in one line:

```
state.x77[,'Life Exp'] > mean(state.x77[,'Life Exp'])
```

```
##        Alabama         Alaska        Arizona       Arkansas
##          FALSE          FALSE          FALSE          FALSE
##     California       Colorado    Connecticut       Delaware
##           TRUE           TRUE           TRUE          FALSE
##        Florida        Georgia         Hawaii          Idaho
##          FALSE          FALSE           TRUE           TRUE
##       Illinois        Indiana           Iowa         Kansas
##          FALSE           TRUE           TRUE           TRUE
```

```
##       Kentucky      Louisiana          Maine       Maryland
##          FALSE          FALSE          FALSE          FALSE
##  Massachusetts       Michigan      Minnesota    Mississippi
##           TRUE          FALSE           TRUE          FALSE
##       Missouri        Montana       Nebraska         Nevada
##          FALSE          FALSE           TRUE          FALSE
##  New Hampshire     New Jersey     New Mexico       New York
##           TRUE           TRUE          FALSE          FALSE
## North Carolina   North Dakota           Ohio       Oklahoma
##          FALSE           TRUE          FALSE           TRUE
##         Oregon   Pennsylvania   Rhode Island South Carolina
##           TRUE          FALSE           TRUE          FALSE
##   South Dakota      Tennessee          Texas           Utah
##           TRUE          FALSE           TRUE           TRUE
##        Vermont       Virginia     Washington  West Virginia
##           TRUE          FALSE           TRUE          FALSE
##      Wisconsin        Wyoming
##           TRUE          FALSE
```

ifelse(): More often, we would like to assign values from different lists based on a vector of TRUE/FALSE values. The `ifelse()` function is a convenient shorthand for this:

```
  # Returns even for 2,4,6, odd for 1,3,5
  ifelse(1:6 %% 2 == 0, "even", "odd")
```

```
## [1] "odd"  "even" "odd"  "even" "odd"  "even"
```

Remember that running smaller sections of R code in the order of operations (`CTRL + R` while that section is highlighted) is a good way to unwrap a more complex line. If the result would be too large to send to the console, str(), head(), or similar functions can be used instead of running the code directly. For the statement above, this would look like the following:

```
  1:6           # A vector from 1 to 6
```

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

```
  1:6 %% 2      # A vector of 0, 1 values
```

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

```
  1:6 %% 2 == 0 # A vector of T/F values
```

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

Note that the values for true and false conditions are vectors themselves. Like other indices in R, their values are recycled if they are shorter than the object they are indexed against. Here, the values for true and false tests are recycled, though not as one might expect:

```
  ifelse(1:6 %% 2 == 0, c("even1", "even2"), c("odd1", "odd2") )
```

```
## [1] "odd1"  "even2" "odd1"  "even2" "odd1"  "even2"
```

2.1.2 For loops

For loops are a common structure for repetitively executing code a specified number of times. In R, loops are defined by the sequence in an iterator variable, which is usually a vector of integers.

```
  for (i in 130:nrow(iris)) {
    cat("The sum of dimensions for plant", i, "is", sum(iris[i,1:4]), "\n" )
  }
```

```
## The sum of dimensions for plant 130 is 17.6
## The sum of dimensions for plant 131 is 18.2
## The sum of dimensions for plant 132 is 20.1
## The sum of dimensions for plant 133 is 17
## The sum of dimensions for plant 134 is 15.7
## The sum of dimensions for plant 135 is 15.7
## The sum of dimensions for plant 136 is 19.1
## The sum of dimensions for plant 137 is 17.7
## The sum of dimensions for plant 138 is 16.8
## The sum of dimensions for plant 139 is 15.6
## The sum of dimensions for plant 140 is 17.5
## The sum of dimensions for plant 141 is 17.8
## The sum of dimensions for plant 142 is 17.4
## The sum of dimensions for plant 143 is 15.5
## The sum of dimensions for plant 144 is 18.2
## The sum of dimensions for plant 145 is 18.2
## The sum of dimensions for plant 146 is 17.2
## The sum of dimensions for plant 147 is 15.7
## The sum of dimensions for plant 148 is 16.7
## The sum of dimensions for plant 149 is 17.3
## The sum of dimensions for plant 150 is 15.8
```

Generally, for loops provide a convenient way to avoid repeating oneself (the DRY principle), and map largely unchanged for users of other languages. However, the R community strongly favors the apply family of commands, which provide a shorter and sometimes more expressive method to execute a similar function across all rows or columns of a matrix, elements of a list, and so on. These we will cover soon, but deserve their own treatment.

2.1.3 While loops

While loops execute while the given condition evaluates as true. These are written like for loops, but with a single conditional statement to evaluate.

```
  z = 1
  while(z < 10) {
    cat("z is", z, "\n")
    z = z + 1
  }
```

```
## z is 1
## z is 2
## z is 3
## z is 4
## z is 5
## z is 6
## z is 7
## z is 8
## z is 9
```

2.1.4 Syntax for flow control and conditional statements

Synactically, single-line conditions can omit braces.

```
  if(TRUE) cat("Condition is true.\n")
```

```
## Condition is true.
```

Special note to those coming from Python: Unlike programming languages with functional whitespace, R requires braces to indicate which statements belong to functions, and which should execute in a loop or conditional statement. Sections that omit these braces will execute only the first line if the condition is true, and continue to execute the next lines independent of the first. Braces are needed to include the entire block of code in a for or while loop,

```
  # Braces omitted
  if(FALSE)
    cat("Condition is false \n")
    cat("Condition is not true. \n")
```

```
## Condition is not true.
```

```
  # Braces included
  if(TRUE) {
    cat("Condition is true. \n")
    cat("Condition is not false. \n")
  }
```

```
## Condition is true.
## Condition is not false.
```

2.2 Functions

R is largely a functional programming language. Most work is organized into circumscribed functions that take several arguments as input, perform a processing task, and return value. A function contains the following components:

1. A name
2. A list of parameters (input)
3. (Sometimes) a return value

Default values can be supplied by assigning them in the function definition, like so. In this example one variable is assigned to NULL, so that we can test whether the user supplied it in the function call.

```
addFunction <- function(a=2, b=NULL) {
  if(!is.null(b)) a + b
  else a + 5
}

addFunction()    # 2 + 5

## [1] 7

addFunction(b=3) # 2 + b

## [1] 5

addFunction(a=3) # 3 + 5

## [1] 8
```

Return values are either implicitly the value in the last statement in the function, or the value contained in a `return()` statement. Usually, this can be a single value or variable. If a function needs to return multiple things, these can be assembled into a list:

```
formatFunction <- function(a=2, b=NULL) {

  data.mat = matrix(1:100, nrow=10, ncol=10)
  anno.mat = data.frame(val1=LETTERS[1:10], val2=LETTERS[11:20])
  patn.mat = list(a = 10, b = 100)

  return(list(data=data.mat, annot=anno.mat, patn=patn.mat))
}
```

```
formatFunction()
```

```
## $data
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,]    1   11   21   31   41   51   61   71   81    91
##  [2,]    2   12   22   32   42   52   62   72   82    92
##  [3,]    3   13   23   33   43   53   63   73   83    93
##  [4,]    4   14   24   34   44   54   64   74   84    94
##  [5,]    5   15   25   35   45   55   65   75   85    95
##  [6,]    6   16   26   36   46   56   66   76   86    96
##  [7,]    7   17   27   37   47   57   67   77   87    97
##  [8,]    8   18   28   38   48   58   68   78   88    98
##  [9,]    9   19   29   39   49   59   69   79   89    99
## [10,]   10   20   30   40   50   60   70   80   90   100
##
## $annot
##    val1 val2
## 1     A    K
## 2     B    L
## 3     C    M
## 4     D    N
## 5     E    O
## 6     F    P
## 7     G    Q
## 8     H    R
## 9     I    S
## 10    J    T
##
## $patn
## $patn$a
## [1] 10
##
## $patn$b
## [1] 100
```

A major aspect of functional programming in R is that function arguments are passed by value, not by reference. In other words, every function argument will end up making a local copy, unless that argument remains unmodified. There are tricks to emulate pass-by-reference semantics in R using workspaces and global variables, though sticking with packages designed for this is often best. In particular, the data.table package avoids copies and passes by reference whenever possible.

Even basic parts of R syntax, like '[' and ':', are defined through functions. If you need to find help on using the subset operator '[', you would use "¿[' " rather than '?['. Alternatively, help for any function in R can be found by highlighting

its name in RStudio and pressing F1.

There are elements of object-oriented programming and inheritance in some R class systems, but in practice these are rarely used.

2.2.1 Getting code for a function

For most functions, the code can be seen by typing the name of the function in the console, without "()". For code that is heavily used, functions are often wrapped into compiled C, which will be indicated by calls to .Internal. These take a bit more work to edit, but for those interested, see Hadley Wickham's tutorials in Advanced R for editing C++ internals using the Rcpp package.

```
  formatFunction
```

```
## function(a=2, b=NULL) {
##
##   data.mat = matrix(1:100, nrow=10, ncol=10)
##   anno.mat = data.frame(val1=LETTERS[1:10], val2=LETTERS[11:20])
##   patn.mat = list(a = 10, b = 100)
##
##   return(list(data=data.mat, annot=anno.mat, patn=patn.mat))
## }
```

2.2.2 Modifying a function

You may need to make minor changes to the way a function works for your project. To do this, the current code can be assigned to a new function, and modified in place. Try making a new version of formatFunction that does not return data to the console.

2.3 R working directory

The working directory is where R looks for data, and the default location to save data and plots. The most common errors in R stem from forgetting to set the working directory. This can be done through menus in most R IDEs. However, it is usually best to set relative working directories explicitly in scripts using setwd().

```
  getwd()
  setwd("C:/Datasets")
```

For larger projects, a project setup script might include relative links to working directories used in downstream analysis scripts. In this way, the project could be re-run from another machine with no major changes.

```r
# Within setup script
data.dir   <- "/Datasets"
result.dir <- "/Results"
report.dir <- "/Reports"
script.dir <- "/Scripts"

# Within analysis script
data <- read.table(file.path(base.dir, data.dir, infile.name))
write.table(result, file=file.path(base.dir, result.dir, outfile.name))
```

2.4 R working environments and memory

The R workspace is where all variables are stored in memory. The entire contents of the workspace can be saved using `save.image()`, which can be handy for saving progress on an intermediate stage in a script. Data in saved this way are kept in a compressed (if desired) binary format that proves much faster than reading and reprocessing the original data. To reload a workspace and included variables with their original names, use the `load()` function on the .Rdata file saved.

2.4.1 Environment namespaces

R functions, data, and variables are defined in the context of a namespace, which helps to avoid collisions between objects. If you inadvertently redefine the concatenation operator c() as a variable name, R will generally continue to use c() properly, because the c() function and c variable are of different classes (and functions cannot take functions as arguments, unlike in Python).

In some cases, functions or variables with the same name will be defined in separate namespaces. R will typically use the one loaded last. To avoid this and use a specific version, you can use '::' to define the namespace in which your variable or function should be found. Find a function that is multiply defined, and use '::' to call a specific version.

For those using recent versions of RStudio, a quick reference to the currently loaded variables, organized by namespace, is under the drop down boxes in the Environment tab. This will likely start loaded with base functions and methods from "stats", "graphics", "methods", etc.

2.4.2 Environment variables

All processing in R in done on variables (or connections) in the working environment. Variables currently in memory can be listed with ls(), and their sizes checked with object.size(). The overall memory consumed, in both RAM

and virtual memory, can be seen in memory.size(). This memory can extend up to a certain predefined limit, which can be checked or changed with memory.limit().

```
  # Check currently loaded variable names
  # optionally, use the pattern argument to collect
  # only those matching a certain naming pattern
  ls(pattern="dir")
```

```
## character(0)
```

```
  # Check the current memory use, in MB
  memory.size()
```

```
## [1] 31.76
```

```
  # Check the current memory limit, in MB
  memory.limit()
```

```
## [1] 16140
```

```
  # Reset the memory limit to a value, in MB
  memory.limit(32768)
```

```
## [1] 32768
```

When good programming practices and efficient algorithms are not enough, the memory limit in R can be raised to a value that includes both the available RAM and maximum size of the virtual memory (page or swapfile).

Since it is commonly useful to list (and which temporary variables are using most of the memory), you may want to define a special version of ls() that makes variable sizes and dimensions more clear, as described in this StackOverflow post.

2.4.3 Garbage collection

R's memory management relies on a garbage collection system. This is normally invoked when available memory is exhausted to clear memory no longer used (for example, from variables that have been removed using rm()), but can be started manually with gc(). Manual garbage collection may be useful before profiling performance, to avoid a large spike for a line it was automatically called at.

Notice, however, that this does not return the working memory to the clean slate that it began with when R was first run: memory errors may still result if a large variable has to be split across multiple smaller available memory chunks, with the typical error "Could not allocate vector of size X Mb". Before taking drastic measures, it may be worth re-running scripts from a fresh instance of R, when memory is available as a more contiguous block.