

# **Part I**

## **Basics**

# Chapter 1

## Set-Up

**Abstract** In this chapter, an introduction to the text as a whole and the basics for getting set-up with the R programming language are given.

### 1.1 Introduction



Exploratory data analysis (EDA), initially described in John Tukey's classic text by the same name, is a general approach to examining data through visualizations and broad summary statistics [6].<sup>1</sup> It prioritizes studying data directly in order to generate hypotheses and ascertain general trends prior to, and often in lieu of, formal statistical modeling. The growth in both data volume and complexity has further increased the need for a careful application of these exploratory techniques. In the intervening 40 years, techniques for EDA have enjoyed great popularity within statistics, computer science, and many other data-driven fields and professions.

Concurrent with Tukey's development of EDA, Rick Becker, John Chambers, and Allan Wilks of Bell Labs began developing software designed specifically for statistical computing.<sup>2</sup> By 1980, the "S" language was released for general distribution outside Labs. It was followed by a popular series of books and updates, including "New S" and "S-Plus" [1, 2, 3, 5]. In the early 1990s, Ross Ihaka and Robert Gentleman produced a fully open-source implementation of S called "R".<sup>3</sup> Their implementation has become the de-facto tool in the field of statistics and is often cited as being amongst the Top-20 used programming languages in the world.<sup>4</sup>

---

<sup>1</sup>An enjoyable and highly recommended biography of Tukey by David Brillinger (a student of his) was recently published by the *Annals of Statistics* [4].

<sup>2</sup>John Tukey in fact split his time between Bell Labs and Princeton during this time.

<sup>3</sup>It is called "R" for it is both the "next letter in the alphabet" and the shared initial in the Authors' names.

<sup>4</sup>See [www.tiobe.com/index.php/tiobe\\_index](http://www.tiobe.com/index.php/tiobe_index) for one such ranking. Exact rankings of programming language use are, however, impossible to produce to everyone's satisfaction, and results often lead to fairly heated debate. Our point is simply to point out that R is not strictly a tool for a small niche of academic research, but is in fact used quite broadly.



There is a clear and strong link between EDA and S/R.<sup>5</sup> Each owes a great deal of gratitude to the other for their continued popularity. Without the interactive console and **flexible graphics engine** of a language such as R, modern data analysis techniques would be largely intractable. Conversely, without the tools of EDA, R would likely still have been a welcome simplification to programming in lower-level languages, but would have played a far less pivotal role in the development of applied statistics.

The historical context of these two topics underscores the motivation for studying both concurrently. In addition, we see this book as contributing to efforts to bring new communities to learn from and to help shape data analysis by offering other fields of study to engage with. It is an attempt to provide an introduction for students and scholars in the humanities and the humanistic social sciences to both EDA and R. It also shows how data analysis with humanities data can be a powerful method for humanistic inquiry.

## 1.2 Structure of This Book

The book is written in two parts. The first half is an overview of R and EDA. Chapter 2 provides a very basic and straightforward introduction to the language itself. Chapters 3–5 give an introduction to the tools of data analysis by way of worked examples using the rich demographic **data** from the United States American Community Survey and the French 2012 Presidential Election. Since concepts and analysis build off work in previous chapters, these introductory chapters are meant to be read sequentially. Chapter 12 provides 100 short programming questions for further practice, with example solutions in Chap. 13.


The second half introduces key areas of analysis for humanities data. Each chapter introduces a type of analysis and how it can be applied to humanities data sets along with practice problems and extensions. While some concepts are referenced between chapters, each is intended to stand on its own (with the exception of Chaps. 9 and 10, which should be read as a pair).

For those new to the concepts covered in this book, we recommend skimming the chapters in the second half that interest you before walking through the first half. The code may look daunting, but do not fret. The second half provides examples and reasons why these methods and forms of analysis are informative and exciting. We find it easier to learn how to code and explore data when we know what scholarly questions and modes of inquiry excite us.

---

<sup>5</sup>We will refer to the language simply as R for the remainder of this text for simplicity and to conform to the majority of other references; this is meant in no way as a lack of appreciation for the historical importance of the original “S”.

## 1.3 Obtaining R

The majority of readers will eventually want to follow along with the code and examples given through the text. The first step in doing so is to obtain a working copy of R. The Comprehensive R Archive Network, or CRAN, is the official home of the R language and supplies download instructions according to a user's operating system (i.e., Mac, Windows, Linux): 

<http://cran.r-project.org/>

A popular alternative, particularly for users with limited to no programming background, is offered by RStudio:

<http://www.rstudio.com/>

The company offers commercial support but provides a single-user version of the software at no cost. Other download options exist for advanced users, up-to and including a custom build from the source code. We make no assumptions throughout this text regarding which operating system or method of obtaining or accessing R readers have chosen. In the rare cases where differences exist based on these options, they will be explicitly addressed.

A major selling point of R is its extensive collection of user-contributed add-ons, called packages. The details of these packages and how to install them are described in detail in Chap. 11.

## 1.4 Supplemental Materials


In addition to the R software, walking through the examples in this text requires access to the datasets we explore. Care has been taken to ensure that these are all contained in the public domain so as to make it easy for us to redistribute to readers. The materials and download instructions can be found here:

<http://humanitiesdata.org/>

For convenience, a complete copy of the code from the book is also provided to make replicating (and extending) our results as easy as possible.

## 1.5 Getting Help with R

Learning to program is hard and invariably questions and issues will arise in the process (even the most experienced users require help with surprisingly high frequency). The first source of help should be the internal R help documentation, which we describe in detail in Chap. 2. When these fail to address a question, a web search will often turn up the desired result.

When further help is required, the R mailing lists, <http://www.r-project.org/mail.html>  and the third-party question and answer site, [stackoverflow.com/](http://stackoverflow.com/), provide mechanisms for submitting questions to a broad community of R users. Both will also frequently show up as highly ranking results when running generic web searches for R help.

## References

- [1] Richard A Becker and John M Chambers. *S: an interactive environment for data analysis and graphics*. CRC Press, 1984.
- [2] Richard A Becker and John M Chambers. *Extending the S system*. Wadsworth Advanced Books and Software, 1985.
- [3] Richard A Becker, John M Chambers, and Allan Reeve Wilks. The new s language: A programming environment for data analysis and graphics, 1988.
- [4] David R Brillinger. John W. Tukey: his life and professional contributions. *Annals of Statistics*, pages 1535–1575, 2002.
- [5] John M Chambers and Trevor J Hastie. *Statistical models in S*. CRC Press, Inc., 1991.
- [6] John W Tukey. Exploratory data analysis. *Reading, Ma*, 231:32, 1977.

# Chapter 2

## A Short Introduction to R

**Abstract** In this chapter, a basic introduction to working with objects in R is given. We provide the minimal working knowledge to work through the remainder of Part I. Basic computations on vectors, matrices, and data frames are shown. Particular attention is given to R's subsetting mechanism as it is often a source of confusion for new users.

### 2.1 Introduction

Here we provide an introduction to the core features of the R programming language. It is meant to provide enough of a background to proceed through Part I. It is not exhaustive, with some important features, such as random variable generation and control flow functions, presented as they are necessary. This chapter should provide the right depth for getting started. Practice problems are also given in Chap. 12 (with solutions in Chap. 13). Anyone comfortable with other scripting languages will most likely find a quick read sufficient. Accordingly, we assume that readers have already managed to download and set up R as described in Chap. 1.

For readers looking for a dense and complete introduction, we recommend the freely available manual “An Introduction to R” [2]. On the other hand, readers looking for a slower introduction to the language may prefer first working through Matthew Jockers’s “Text Analysis with R for Students of Literature” [1].

### 2.2 Calculator and Objects




The R console can be used as a simple calculator. Typing in a mathematical expression and hitting enter prints out the result.

```
> 1 + 2
[1] 3
```

The familiar **order of operation rules** worked as expected and **many mathematical functions** such as the square root, `sqrt`, can also be applied.

```
> 1 / (2 + 17) - 1.5 + sqrt(2)
[1] -0.03315486
```

The result of a mathematical expression can be assigned to an object in R using the **<-**  operator.

```
> x <- 1 + 2
```

When assignment is used, the result is no longer printed in the console window. If we want to see the result, we type the variable name `x` to print out its value.

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

We can apply further manipulations to the constructed objects; for example, here we divide the value of `x` by 2.


```
> x/2
[1] 1.5
> x
[1] 3
```



The result 1.5 prints to the console; however, notice that the actual value of `x` has not changed. If we want to save the output it has to be reassigned to a variable (this does not have to be a new variable; `x <- x/2` is allowed). Here we construct an object named `y`.

```
> y <- x/2
> y
[1] 1.5
```

At this point the object named `x` has a value of 3 and the object named `y` has a value of 1.5.

Every object in R belongs to a *class* describing the type of object it represents. To determine an object's class, we use a function called `class`<sup>1</sup>; this function has one input parameter named `x`. Accordingly, this input can be explicitly called (`class(x=y)`) or implicitly (`class(y)`).

```
> class(x=y) 
[1] "numeric"
> class(y)
[1] "numeric"
```

<sup>1</sup>Functions are reusable code. Functions have a set of  mptions built in and can be accessed by placing a `?` before the function name. Type `?class()`  R console. Note the usage information. It provides information about the functions' inputs. This particular function has one input parameter named `x`. Scroll down and inspect the documentation about the function. It may seem overwhelming at first, but it will be an important tool. Type `q` to exit and return to coding.

In the second case, R assumes that since we did not name the input we intended the variable `y` to be assigned to the first (and in this case only) input `x`. The result shows that `y` is described as an object of type `numeric`; this is a generic class that holds any type of real number.

Everything in R is an object, including functions. Therefore we can even pass the function `class` to itself.

```
> class(x=class)
[1] "function"
```



We see here that “function” is a type of class. Another useful basic function is `ls` (LiSt objects). It does not require any arguments, and prints out the names of all the objects we have created.

```
> ls()
[1] "x" "y"
```



Notice that this list does not include the function `class`, even though we have determined that it exists and is a type of object. The reason is that by default only user-created objects are returned. If we really wanted to see all of the objects created in the base of R, like `class`, we need to override one of the default parameters of the function `ls`.<sup>2</sup>

```
> ls(envir=baseenv())
...
[303] "chol.default"           "chol2inv"
[305] "choose"                 "class"
[307] "class<-"                "clearPushBack"
[309] "close"                  "close.connection"
...
```



The output is quite long, so we only display the few lines where the function `class` is shown. We show this as an example of the power in R’s function syntax; allowing functions to have default values makes it easy to get simple results while not limiting the ability of users to customize functions as necessary.

## 2.3 Numeric Vectors



In R, a **vector** is a **data structure** containing a collection of multiple values with the **same type**. There are several types of vectors, six to be exact, with one of the most common being a numeric vector. A special function `c(...)` combines multiple values into a single vector object. For example, to create an object with the values 1, 10, and 100 run the following.

<sup>2</sup>We suggest trying this on your machine as well (do not worry about fully understanding the function call) in order to get an appreciation of the number of functions which are available by default within the base R language.



```
> vecObj <- c(1,10,100)
> vecObj
[1] 1 10 100
> class(vecObj)
[1] "numeric"
```

The resulting object is referred to as a vector. When we determine the object's class, we see that it is still a “numeric” object just like the single number objects explored in the previous section. The reason for this is that in R a single numeric value is represented as a length one vector rather than a separate type.<sup>3</sup>

We can conduct mathematical manipulations directly on vectors. Consider the following two examples:

```
> vecObj + 10
[1] 11 20 110
> vecObj + vecObj
[1] 2 20 200
```

In the first, we took a vector of length 3 and added a single number to it. The result added the single number to each element in the vector. In the second example, we add together two vectors each of length 3, with a result that adds each element of the vectors. In general, R evaluates expressions involving vectors of different lengths by **recycling the shorter ones to match the longest one**. For instance, if we add a length 6 vector to a length 2 vector, the first element of the shorter vector is added to the 1st, 3rd, and 5th elements of the longer one, whereas the second element of the shorter vector is added to the 2nd, 4th, and 6th elements of the longer one.

```
> c(1,2,3,4,5,6) + c(100,200)
[1] 101 202 103 204 105 206
```



The most common case of manipulating vectors involves expressions that mix vectors of a single given length with length-one vectors, but is important to recognize the more general case.

Constructing vectors by hand can quickly become cumbersome. A shortcut for building a vector of all the integers between two numbers is the colon operator, “:”.<sup>4</sup> For example `1:84` returns a vector of all integers between 1 and 84.

```
> 1:84
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
[22] 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42
[43] 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
[64] 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84
```



<sup>3</sup>The lack of a *scalar* type in R to represent individual objects is a major departure from many other programming languages such as Python, C, and Java.

<sup>4</sup>In R there is a formal distinction between *integer* (“whole” numbers) and *numeric* objects. However, for the level of this text, the distinction will not be important. R will silently convert between the two as necessary and should never cause unexpected behavior due to the distinction. We will use the term “integer” throughout to describe something which should be a whole number, but may be formally represented by an object of class *numeric*.

Finally, it is often useful to know the length of a given vector. In order to determine this, the `length` function is used. For example, `length(x=10:42)` would return the number 33.

## 2.4 Logical Vectors

Another useful and common type of vectors in R are logical vectors, which can be constructed as the output of using the expressions `>` (greater than), `>=` (greater than or equal), `<` (less than), `<=` (less than or equal), `==` (equal), and `!=` (not equal).

```
> numericVec <- 1:10
> logicalVec <- (numericVec >= 5)
> logicalVec
[1] FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE TRUE
> class(logicalVec)
[1] "logical"
> numericVec == 4
[1] FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE
```

In addition to creating logical vectors by manipulating numeric ones, we can also construct vectors by using the symbols `TRUE` and `FALSE`.

```
> logicalVec <- c(TRUE, TRUE, FALSE, TRUE)
> class(logicalVec)
[1] "logical"
```

Logical vectors can be manipulated via the `!` (not), `|` (or), and `&` (and) expressions.

```
> logicalVec1 <- c(FALSE, FALSE, TRUE, TRUE)
> logicalVec2 <- c(FALSE, TRUE, FALSE, TRUE)
> logicalVec1 | logicalVec2
[1] FALSE TRUE TRUE TRUE
> logicalVec1 & logicalVec2
[1] FALSE FALSE FALSE TRUE
> !logicalVec1
[1] TRUE TRUE FALSE FALSE
```

Finally, logical vectors are converted to numeric ones when appropriate by mapping `TRUE` to the number 1 and `FALSE` to the number 0. Examples include using logical vectors in mathematical expressions.

```
> logicalVec1 + logicalVec2
[1] 0 1 1 2
> logicalVec1 + 1
[1] 1 1 2 2
> class(logicalVec1 + 1)
[1] "numeric"
```

We shall see that logical vectors are useful tools for filtering and manipulating other R objects.

## 2.5 Subsetting

There are four basic methods for accessing a subset of an R vector. We will discuss the three most common here. Constructing subsets of vectors in R is straightforward in principle, but it does quickly lead to fairly intricate, and often complex, code.

The syntax for all subsetting commands uses square brackets, `[` and `]`, immediately following the name of the vector. If an integer  $i$  is placed in-between the brackets the element in position  $i$  is returned.<sup>5</sup> Here, for example, we pull out the 20th element of the vector `vectorObj`.

```
> vectorObj <- 101:132
> vectorObj[20]
[1] 120
```

The integer index notation can also use an assignment. Here is an example of setting the 17th element to  $-2$ :

```
> vectorObj[17] <- -2
> vectorObj
 [1] 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116
[17]  -2 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132
```

The integer input inside of the brackets need not be of length one (remember that “numbers” are just vector of length one). We use the same notation to extract four elements of the vector; notice the usefulness of the colon operator for accessing a consecutive sequence of a vector.

```
> vectorObj[c(6,17,20,30)]
[1] 106  -2 120 130
> vectorObj[5:10]
[1] 105 106 107 108 109 110
```

Again, the same notation can be used to assign multiple values to a vector.

The integer subsetting mechanism provides some powerful operations that may not be immediately obvious. There is no restriction on how often an element can be returned. For example, we can create a length three vector where every element is equal to element 20 from our vector `vectorObj`.


```
> vectorObj[c(20,20,20)]
[1] 120 120 120
```

Or, by using an index that runs from the length of the vector down to 1, the reverse of the vector will be returned.

```
> vectorObj[length(vectorObj):1]
[1] 132 131 130 129 128 127 126 125 124 123 122 121 120 119 118  -2
[17] 116 115 114 113 112 111 110 109 108 107 106 105 104 103 102 101
```




<sup>5</sup>Note that R starts indexing at 1 instead of 0 unlike many other languages (Python, Java, C, etc.).

The concept of **recycling**  values from vector arithmetic can also be applied to subset assignment. For example, we can assign the number 1 to the first 10 elements of the vector.

```
> vectorObj[1:10] = 1
> vectorObj
[1] 1 1 1 1 1 1 1 1 1 1 111 112 113 114 115 116
[17] -2 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132
```


The single value was recycled to set the first 10 elements to 1.

The **second**  method for subsetting a vector uses a vector of **negative integers** **between 1 and 17** in square brackets resulting in a vector with the corresponding elements removed.

```
> vectorObj <- 101:132
> vectorObj[-1]
[1] 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117
[17] 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132
> vectorObj[-c(1,length(vectorObj)))]
[1] 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117
[17] 118 119 120 121 122 123 124 125 126 127 128 129 130 131
```


This method is the least likely to be complex as the result is a true subset of the original vector without any duplication or permutation of the results.

The final method for subsetting vectors shown here uses a logical vector in between the square brackets. Only elements corresponding to TRUE are returned.

```
> vectorObj <- 1:8 
> vectorObj[c(TRUE, TRUE, FALSE, FALSE, FALSE, TRUE, TRUE, FALSE)]
[1] 1 2 6 7
```

Typically the logical vector will have the same length of the original vector; when this is not the case, elements of the logical index are recycled.

**The most common usage of logical subsetting** involves constructing the logical vector by inequalities on the original vector. For example, the following code snippet returns the elements of `vectorObj` that have elements greater than 5.

```
> logicalIndex <- vectorObj > 5 
> logicalIndex
[1] FALSE FALSE FALSE FALSE TRUE TRUE TRUE
> newVectorObj <- vectorObj[logicalIndex]
> newVectorObj
[1] 6 7 8
```

As with the previous two subsetting commands, logical indices can be used for object assignment. Here we take a vector of integers between 1 and 100 and truncate those values that are less than 25 and greater than 75.

```

> vectorObj <- 1:100
> vectorObj[vectorObj <= 25] <- 25
> vectorObj[vectorObj >= 75] <- 75
> vectorObj
 [1] 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25 25
[22] 25 25 25 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42
[43] 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
[64] 64 65 66 67 68 69 70 71 72 73 74 75 75 75 75 75 75 75 75 75
[85] 75 75 75 75 75 75 75 75 75 75 75 75 75 75 75

```

Notice that we did not explicitly construct and save a logical vector. Instead the expressions `vectorObj <= 25` and `vectorObj >= 75` were directly passed inside the square brackets; the indexing vector was constructed “on the fly” and thrown away at the end. This is a common paradigm in R scripts for taking subsets of numeric vectors.

## 2.6 Character Vectors

In addition to numeric and logical vectors, R has a class for character vectors.<sup>6</sup>

```

> stringVec <- c("pear", "apple", "pineapple")
> class(stringVec)
[1] "character"
> stringVec
[1] "pear"      "apple"     "pineapple"

```

Many functions are available specifically for working with character vectors. For example, the `paste` function combines two or more character vectors. Here we paste the word “juice” onto the character vector of fruits. Because of the recycling logic in R, “juice” only needs to be typed once rather than three times. We do need to make sure to indicate the separator by using `sep = " "`. In this case, a space is desired.

```

> paste(stringVec, "juice", sep = " ")
[1] "pear juice"      "apple juice"     "pineapple juice"

```



The `substr` command returns substrings of the inputs by using the character offsets. Here we take the second, third, and fourth characters from every element in the vector:

```

> > substr(stringVec, start=2, stop=4)
[1] "ear" "ppl" "ine"

```

The parameters `start` and `stop` can be passed inputs, which are the same length as the string in order to take different subsets of each element. This can be used together with the `nchar` function which returns the number of characters in each

<sup>6</sup>As mentioned earlier, there are formally six primitive types of vectors, but we will not need to use *raw* or *complex* vectors. As well, we do not need to distinguish between *numeric* and *integer* vectors.

element of the character vector to return a set of substrings that removes just the first two characters of each string.

```
> nchar(stringVec)
[1] 4 5 9
> substr(stringVec, 3, nchar(stringVec))
[1] "ar"      "ple"      "neapple"
```

The function `grep` takes a pattern and returns the indices that have characters containing the given pattern. These indices can be used with the subsetting commands to return a subset of a character vector corresponding to those elements containing the particular pattern. For example, if we use the pattern “apple” in the `grep` function, the elements “apple” and “pineapple” can be extracted.

```
> index <- grep(pattern="apple", x=stringVec)
> index
[1] 2 3
> stringVec[index]
[1] "apple"      "pineapple"
```

The pattern element to `grep`, which stands for global **regular expression** print, need not be a fixed string and may contain wildcard characters such as `*`. The input can in fact be any pattern corresponding to a *regular expression*; see the help page `?regexpr` for a complete description.

We have already seen that logical vector gets converted to numeric ones when we try to use them in arithmetic. What happens when we try to use a character vector that contains a number in quotes in addition?

```
> "1" + 1
Error in "1" + 1 : non-numeric argument to binary operator
```

An error gets thrown refusing to use character vectors in arithmetic operations. However, if we try to use a numeric vector in a character operation such as `paste`, the command runs without a problem by silently converting the numeric values to characters.

```
> paste(4:12, "th", sep = "")
[1] "4th" "5th" "6th" "7th" "8th" "9th" "10th" "11th" "12th"
```

The reason for this is that R has the concept of **implicit type coercion**. Accordingly, objects are automatically converted to the appropriate type, but only when this can be **done unambiguously and without any possible errors**. Generally, this means that logical and numeric vectors are converted between one another and into character vectors whenever needed; however, character vectors must be explicitly cast into numeric objects because this may cause errors when a string does not actually represent a number (such as “pear”).

The function `as.numeric` is used to cast any vector into a numeric one; it can be used to fix our previous error:

```
> as.numeric("1") + 1
[1] 2
```



Other casting functions such as `as.logical` and `as.character` can be used to explicitly convert vectors into their respective types.

## 2.7 Matrices and Data Frames

Matrices (plural form of matrix) provide an extension to vectors by providing a dimensionality to the data. A matrix object can be constructed from a given vector by providing the desired number of rows and columns. The resulting object, when printed, displays the data in a grid with the given number of columns and rows.

```
> mat <- matrix(data=1:12, nrow=3, ncol=4, byrow=FALSE)
> mat
      [,1] [,2] [,3] [,4]
[1,]     1     4     7    10
[2,]     2     5     8    11
[3,]     3     6     9    12
> class(mat)
[1] "matrix"
```

The option `byrow` determines whether the vector fills up the matrix over rows or columns.

```
> mat <- matrix(data=1:12, nrow=3, ncol=4, byrow=TRUE)
> mat
      [,1] [,2] [,3] [,4]
[1,]     1     2     3     4
[2,]     5     6     7     8
[3,]     9    10    11    12
> class(mat)
[1] "matrix"
```

To access a given element, the same bracket notation is used except that now two sets of indices are given and separated by a comma with the first number denoting the desired rows and the second giving the columns. Any of the subsetting vectors (integers, negative integers, and logical vectors) can be used.

```
> mat[2,3]
[1] 7
> class(mat[2,3])
[1] "integer"
> mat[1:2,1:2]
      [,1] [,2]
[1,]     1     2
[2,]     5     6
> class(mat[1:2,1:2])
[1] "matrix"
```

In the case of only wanting to subset by one dimension, the other dimension's index can **simply be left blank (understood to mean all rows/columns)**. For example,

grabbing all rows from the 2nd and 3rd columns uses `[, 2:3]` as the subsetting command.

```
> mat[,2:3]
      [,1] [,2]
[1,]     2     3
[2,]     6     7
[3,]    10    11
```

Care should be taken when extracting only a single row or column, as by default R will convert the matrix into a vector. The additional parameter `, drop=FALSE` can be passed inside the brackets to stop this behavior.

```
> class(mat[1,])
[1] "integer"
> class(mat[1,, drop=FALSE])
[1] "matrix"
```

The automatic demotion to vectors is a common cause of subtle bugs in R scripts.

It is also possible to use matrices in arithmetic calculations. When used in combination with vectors, the vector values are recycled throughout the length of the matrix.<sup>7</sup> Basic operations between matrices require each to have the same dimensions and the operations are applied element-wise.

```
> matCol <- mat[2:3, ](data=1:12, nrow=3, ncol=4, byrow=FALSE)
> matCol + 1
      [,1] [,2] [,3] [,4]
[1,]     2     5     8    11
[2,]     3     6     9    12
[3,]     4     7    10    13
> mat + matCol
      [,1] [,2] [,3] [,4]
[1,]     2     6    10    14
[2,]     7    11    15    19
[3,]    12    16    20    24
> mat + matCol[,2:3]
Error in mat + matCol[, 2:3] : non-conformable arrays
```

It is possible to construct matrices from any vector class (such as logical, numeric, or character), though by far the most common is the numeric case.

In many cases, it is advantageous to have a matrix-like object where each column has a different data type. The *data frame* object was created for exactly this purpose; the prevalence of such an object in the base language of R is a result of the language's history as a tool for statistics and data analysis. To construct a data frame, the function *data.frame* is used with each desired column of data passed as a separate argument; these are usually named in order to denote the meaning of each column. Here we construct a data frame with three columns named "a", "b", and "c".

<sup>7</sup>A warning is given if the vector's length does not divide the length of the matrix; an error is thrown when the vector is longer than the matrix.



```
> df <- data.frame(a = 1:5, b=21:25, c=1:5 + 0.5)
> df
  a  b  c
1 1 21 1.5
2 2 22 2.5
3 3 23 3.5
4 4 24 4.5
5 5 25 5.5
> class(df)
[1] "data.frame"
```

Three useful properties called **attributes** are attached to every data frame object in R: the **dimension** of the data frame, the **column names**, and the **rownames**. These are accessed as follows:

```
> dim(df)
[1] 5 3
> colnames(df)
[1] "a" "b" "c"
> rownames(df)
[1] "1" "2" "3" "4" "5"
```

The dimension command returns a vector with the number of rows as the first element and number of columns as the second. The vectors resulting from these three commands can also be changed by assigning to them. For example, the following changes the second variable name from “b” to “newName”.


```
> colnames(df)[2] <- "newName"
> df
  a newName  c
1 1      21 1.5
2 2      22 2.5
3 3      23 3.5
4 4      24 4.5
5 5      25 5.5
```

**These three commands also work for matrices.** The output of dim for matrices is in exactly the same format as for data frames; the column names and row names are slightly different because by default matrices have missing names (data frames always have these). However, these can be set and manipulated manually in the same way.

The matrix subsetting commands work the same way on data frames. There is also an additional (and very useful) way to access a single column of a data frame by using the **\$ operator** followed by the variable name.

```
> df$newName
[1] 21 22 23 24 25
> class(df$newName)
[1] "integer"
```

The dollar sign notation can also be used to construct a new variable attached to a data frame. Any referenced variable will be constructed at the end of the data frame when referenced.



```
> df$newColumn <- 5:1
> df
  a newName    c newColumn
1 1      21 1.5         5
2 2      22 2.5         4
3 3      23 3.5         3
4 4      24 4.5         2
5 5      25 5.5         1
```

The dollar sign notation does not work for matrices, even when column names are manually constructed.

## 2.8 Data I/O

Beyond typing vectors, matrices, and data frames directly, we can also load external data into R. As a prelude to this, we will need to interact with the computer system. At any given point, R has a notion of a **current working directory**; this is a location in the file system where it will read and write inputs and outputs. The default location will depend on your specific operating system and the method by which you are accessing R (i.e., RStudio, the console, an executable, or the terminal).

The current working directory is displayed with the **getwd** function and can be changed via the **setwd** function. For example, the following code snippet changes the working directory from a user's home to their desktop on Mac OSX.

```
> getwd()
[1] "/Users/myUserName"
> setwd("/Users/myUserName/Desktop")
> getwd()
[1] "/Users/myUserName/Desktop"
```

On Windows, here is an example of the same process changing from a users' home directory to their documents directory.

```
> getwd()
[1] "C:/Users/myUserName"
> setwd("C:/Users/myUserName/Documents")
> getwd()
[1] "C:/Users/myUserName/Documents"
```

Notice that we have used the forward slash / rather than the more typical to Windows backslash \; this is because R recognizes the backslash as an escape character and would otherwise throw an error.<sup>8</sup>

---

<sup>8</sup>It is also possible to use double backslashes on Windows machines. We recommend the forward slash as it makes code cross-compatible between operating systems.

The first step to reading in a dataset is to set the working directory to the location containing the data. Once there, the `dir` function displays a character vector of all the files in the current working directory. Here we have navigated to a directory containing two small files regarding properties of fruit.

```
> dir()
[1] "fruitData.csv"      "fruitNutrition.csv"
```

The extensions “csv” indicate that these are *comma separated value* files; data is written in a tabular form in plain text with rows separated by newline characters and columns separated by commas. Files in this format can be exported from various database and spreadsheet programs. To read one of these files use the `read.csv` function and save the result as an R object.<sup>9</sup> Note that we have set the option `as.is=TRUE`. We will do this regularly for loading and manually constructing data frames. It is necessary in order to stop R from constructing factors as in general we will not be using them.<sup>10</sup>

```
> fruitData <- read.csv(file="fruitData.csv", as.is=TRUE)
> fruitData
  Fruit Color Shape Juice
1 apple   red  round  1.0
2 banana yellow oblong  0.0
3 pear   green  pear   0.5
4 orange orange round   1.0
5 kiwi   green  round   0.0
> class(fruitData)
[1] "data.frame"
> class(fruitData$Juice)
[1] "numeric"
```

The output is a data frame containing five rows and four columns. Notice that in reading the data, R has detected that the column named “Juice” should be stored as a numeric variable.<sup>11</sup>

An analogous function `write.csv` exists for saving data in R as a csv file. Consider saving only the “Juice” variable as a new plain text file.

```
> write.csv(x=fruitData$Juice, file="fruitDataJuice.csv")
```

The output will be saved in the current working directory. Opening the resulting file in a text editor shows that more than just the five Juice numbers have been saved.

<sup>9</sup>The function `read.csv` is a shortcut to the more general function `read.table`, which has over a dozen options for reading in a number of plain text formats.

<sup>10</sup>Factors are mainly used for statistical modeling. Even in those cases, most avoid them until absolutely necessary. As a result, we too will avoid factors as they are more likely to cause complications than help with humanities data.

<sup>11</sup>Juice = 1 indicates fruits commonly sold in juice form in the U.S. and Juice = 0.5 to those sometimes available in juice form.

```
"", "x"
"1", 1
"2", 0
"3", 0.5
"4", 1
"5", 0
```

The additional data on the first line and the first column are there to represent row and column names. Prior to saving the `fruitDataJuice` vector, it was coerced to a data frame. In the process, default row and column names were attached (recall that data frames always have these). Reading the data back into R shows that the `fruitDataJuice` data is now saved as a data frame object.

```
> fruitDataJuice <- read.csv(file="fruitDataJuice.csv")
> class(fruitDataJuice)
[1] "data.frame"
```

We could have avoided the additional row and column names from existing in the plain text csv file by specifying additional parameters in the function call, but there is no way of using `read.csv` to output anything other than a data frame.

An alternative way of saving R data is to save a serialized object as an R data file. The extension `“.rds”` is commonly used for the output.

```
> saveRDS(object=fruitData$Juice, file="fruitDataJuice.rds")
```

Opening this in a text editor shows an uninterpretable jumble of characters. It will look different in different text editors, all of which will be unreadable. Below is one example.

```
??b'`b'fdb'b2?????
????1??>
```

However, reading it back into R is a simple matter of calling the `readRDS` function.

```
> fruitDataJuice <- readRDS(file="fruitDataJuice.rds")
> class(fruitDataJuice)
[1] "numeric"
> fruitDataJuice
[1] 1.0 0.0 0.5 1.0 0.0
```

In this case, the result was returned exactly as we saved it: a numeric vector.

As a general rule of thumb, plain text files such as csv are best used when sharing data with others or when storing the final results of an analysis. These files are much easier to understand when looking at the output and can be easily imported into other software and programs. On the other hand, R data files are great for storing intermediate results as they can be saved and written without worrying about losing details in the conversion process. **When dealing with larger datasets, R data files are beneficial because they take less storage space on the disk and can be loaded back into R significantly faster.**



## 2.9 Advanced Subsetting

There are several functions in R written specifically in order to assist with complex forms of subsetting. Consider the task of taking only those rows in our fruit data frame with color equal to “red” or “green”. One method based on our current tools constructs a logical vector using the “or” (`|`) operator.

```
> index <- (fruitData$Color == "red" | fruitData$Color == "green")
> index
[1] TRUE FALSE TRUE FALSE TRUE
> fruitData[index,]
  Fruit Color Shape Juice
1 apple  red round  1.0
3 pear green  pear  0.5
5 kiwi green round  0.0
```

This can quickly become cumbersome when the number of categories becomes large. The `%in%` operator provides a convenient shortcut for constructing the logical index vector.

```
> fruitData$Color %in% c("red", "green")
[1] TRUE FALSE TRUE FALSE TRUE
```

Now that we have identified the red and green fruits, we can pull out the corresponding rows. Remember that when we subset a data frame, we need to specify both rows and columns. Because we want all the columns, we will leave a blank after the comma inside of the square brackets.

```
> fruitData[fruitData$Color %in% c("red", "green"),]
  Fruit Color Shape Juice
1 apple  red round  1.0
3 pear green  pear  0.5
5 kiwi green round  0.0
```

Using this, for instance, we can quickly discover that only one fruit in our data has a corresponding color named after it.

```
> fruitData[fruitData$Color %in% fruitData$Fruit,]
  Fruit Color Shape Juice
4 orange orange round  1
```

Writing without the `%in%` operator, this would have required a much longer chain of conditional logic.

Another common task is to order a data frame based on a single column. The function `order` returns the ordering of integer indices that would sort the vector. By default this is done in ascending order, but can be reversed with the `decreasing` option. The indices can be used to then reorder a vector, matrix, or data frame.

```
> index <- order(fruitData$Juice, decreasing=TRUE)
> index
[1] 1 4 3 2 5
> fruitData[index,]
  Fruit Color Shape Juice
1  apple   red  round  1.0
4  orange orange round  1.0
3   pear green  pear  0.5
2 banana yellow oblong  0.0
5   kiwi green  round  0.0
```

When presented with ties, as was the case with the Juice variable, the `order` function uses a stable sort so that the relative ordering is preserved. For example, index 1 (“apple”) came before index 4 (“orange”) because they had the same value and 1 is less than 4.

Often data analysis involves combining independent data sets. Consider a data frame relating fruits to their caloric content.

```
> fruitNutr <- read.csv("fruitNutrition.csv", as.is=TRUE)
> fruitNutr
  Fruit Calories
1 banana     100
2  pear     100
3  mango     200
```

In order to combine this data to our original data, we need to know how to relate the two datasets. The function `match` takes two vectors and, according to the R help documentation, “returns a vector of the positions of (first) matches of its first argument in its second”. The result can be used to join the two datasets together. However, in our case, not all of the fruits in our original set have nutritional data, so the `match` function returns the values `NA` for the other three fruits.

```
> index <- match(x=fruitData$Fruit, table=fruitNutr$Fruit)
> index
[1] NA  1  2 NA NA
```

These are *missing values*, the abbreviation means “not available”. To deal with these, the function `is.na` returns a logical vector of the positions with missing values.<sup>12</sup>

```
> is.na(index)
[1] TRUE FALSE FALSE  TRUE  TRUE
```

In order to join these datasets, we need a new column to insert the calories data. Therefore, we set the calories variable in the original dataset to missing.

```
> fruitData$Calories <- NA
```

---

<sup>12</sup>You may be tempted to try `index == NA`. It will run without an error but not return the desired result.

Then, use a combination of the `is.na` function and the variable `index`.

```
> fruitData$Calories[!is.na(index)] <-  
+ fruitNutr$Calories[index[!is.na(index)]]  
> fruitData
```

	Fruit	Color	Shape	Juice	Calories
1	apple	red	round	1.0	NA
2	banana	yellow	oblong	0.0	100
3	pear	green	pear	0.5	100
4	orange	orange	round	1.0	NA
5	kiwi	green	round	0.0	NA

The data frame `fruitData` now contains the caloric count for the two matching fruits.

## References

- [1] Matthew Lee Jockers. *Text Analysis with R for Students of Literature*. Springer, 2014.
- [2] William N Venables, David M Smith, R Development Core Team, et al. An introduction to r, 2002.

## Chapter 3

# EDA I: Continuous and Categorical Data

**Abstract** In this chapter, basic methods for exploratory data analysis are presented. The focus is on univariate and bivariate graphical techniques such as histograms, bar plots, and tables. Control flow using *for loops* is also introduced.

### 3.1 Introduction

The R programming language was originally intended as a tool for *exploratory data analysis*. In the previous chapter we covered the basic constructs of the language; here we move into the more data-specific functionality of R. The focus will slowly shift away from the language itself, which is ultimately just a tool, and towards the conceptual methods used for exploring data.

Here, and again in Chap. 4, we use the American Community Survey (ACS) as an example dataset [1]. The ACS is produced annually by the US Census Bureau. Unlike the decennial census, it is a survey of only approximately 1 % of the population, but asks a substantially longer and broader set of questions. This survey is convenient to work with as it is in the public domain and freely downloadable. The variables in the survey, such as age, income, and occupation, are all interpretable without specialized subject domain knowledge. Also, the ACS, and demographic data in general, is likely to be of direct interest and application within a wide range of academic disciplines in both the humanities and social sciences. Due to its size, we have cleaned the data. We will only look at tract-level data from the state of Oregon. Additional states and tables are provided in the supplementary materials for further study.



## 3.2 Tables

We start by loading the base ACS file into R using the `read.csv` function.

```
> geodf <- read.csv("data/ch03/geodf.csv", as.is=TRUE)
> dim(geodf)
[1] 826 6
```

Printing the first six rows reveals six variables in this dataset.

```
> geodf <- read.csv("data/ch03/geodf.csv", as.is=TRUE)
> dim(geodf)
[1] 826 6
> geodf[1:6,]
  state   fips county  csa population households
1  or 1950100 Baker None    2725      1225
2  or 1950200 Baker None    3179      1322
3  or 1950300 Baker None    2395      1162
4  or 1950400 Baker None    2975      1397
5  or 1950500 Baker None    2969      1117
6  or 1950600 Baker None    1812       897
```

Each row represents a *census tract*, a county subdivision used for aggregating statistical survey data. They typically correspond to individual towns or small cities, though in larger metropolitan areas tracts more closely represent specific neighborhoods.

Looking closer at the columns, the state variable will not be of much help as all of the data came from Oregon. The fips code is a unique identifier of each row. The county and csa, short for Combined Statistical Area, fields give larger-scale groupings of individual census tracts. The population and household columns are estimated values from the survey sample. Accordingly, all of the data counts we will look at are similarly “scaled-up” from the 1 % sample taken during the survey.

The first few tracts represented by rows of the dataset belong to Baker County. How many counties are there in total in our dataset? Does each county have roughly the same number of tracts? To answer these question, we use the `table` function, which takes a character vector and provides a count for each of the unique values in the vector.<sup>1</sup>

```
> tab <- table(geodf$county)
> class(tab)
[1] "table"
> tab
```

Baker	Benton	Clackamas	Clatsop	Columbia	Coos
6	18	80	11	10	13
Crook	Curry	Deschutes	Douglas	Gilliam	Grant
4	5	24	22	1	2
Harney	Hood River	Jackson	Jefferson	Josephine	Klamath
2	4	41	6	16	20

<sup>1</sup>Recall that numeric and logical vectors can be coerced to characters when needed; the table function is able to tabulate numeric variables using this trick.

Lake	Lane	Lincoln	Linn	Malheur	Marion
2	86	17	21	7	58
Morrow	Multnomah	Polk	Sherman	Tillamook	Umatilla
2	171	12	1	8	15
Union	Wallowa	Wasco	Washington	Wheeler	Yamhill
8	3	8	104	1	17

The output of the table function is a new object type “table”, **though it can be manipulated exactly like any named vector for our purposes**. From the results, we see a set of 36 counties, with some (Grant, Wheeler, Wasco) having only one tract and others (Multnomah, Washington) having over 100.

**By default, a table of a character vector will arrange the results in alphabetical order.** To arrange the results based on the frequency counts, we evoke the same order function and subsetting commands that were previously used to order numeric vectors. We can also use an additional parameter `decreasing=TRUE` in the function to order the table; the result is the county with the most census tracts.

```
> tab[order(tab, decreasing=TRUE)]
```



Multnomah	Washington	Lane	Clackamas	Marion	Jackson
171	104	86	80	58	41
Deschutes	Douglas	Linn	Klamath	Benton	Lincoln
24	22	21	20	18	17
Yamhill	Josephine	Umatilla	Coos	Polk	Clatsop
17	16	15	13	12	11
Columbia	Tillamook	Union	Wasco	Malheur	Baker
10	8	8	8	7	6
Jefferson	Curry	Crook	Hood River	Wallowa	Grant
6	5	4	4	3	2
Harney	Lake	Morrow	Gilliam	Sherman	Wheeler
2	2	2	1	1	1

The rearranged table makes our observations from the original table more obvious; the advantage becomes more noticeable with larger tables. Notice that the table has reordered both the names and the counts together. To access just the counts we can convert the table to a numeric vector via `as.numeric`. Accessing just a character vector of names requires calling the function `names`, a variation of the `colnames` and `rownames` we used when working with data frames and matrices.<sup>2</sup> In order to get the names of the five counties with the highest number of tracts, we use the following code snippet:

```
> names(tab)[order(tab,decreasing=TRUE)][1:5]
[1] "Multnomah" "Washington" "Lane" "Clackamas" "Marion"
```

We will see that this is a useful way of extracting values from character vectors with a large number of unique values.

The `table` function is not restricted to creating univariate tables with just one input. It can be used to tabulate an arbitrary number of inputs. For example, the

<sup>2</sup>The `names` function should not be used with matrices; for data frames it returns the column names.

number of counties in each combined statistical area (CSA) might be of interest. To understand the relationship between the CSA and counties within Oregon, we build a two-way table of these two variables.<sup>3</sup>

```
> table(geodf$county, geodf$csa)
```

	Bend	Medford	None	Portland
Baker	0	0	6	0
Benton	0	0	18	0
Clackamas	0	0	0	80
Clatsop	0	0	11	0
Columbia	0	0	0	10
Coos	0	0	13	0
Crook	4	0	0	0
Curry	0	0	5	0
Deschutes	24	0	0	0
Douglas	0	0	22	0
Gilliam	0	0	1	0
Grant	0	0	2	0
Harney	0	0	2	0
Hood River	0	0	4	0
Jackson	0	41	0	0
Jefferson	0	0	6	0
Josephine	0	16	0	0
Klamath	0	0	20	0
Lake	0	0	2	0
Lane	0	0	86	0
Lincoln	0	0	17	0
Linn	0	0	21	0
Malheur	0	0	7	0
Marion	0	0	58	0
Morrow	0	0	2	0
Multnomah	0	0	0	171
Polk	0	0	12	0
Sherman	0	0	1	0
Tillamook	0	0	8	0
Umatilla	0	0	15	0
Union	0	0	8	0
Wallowa	0	0	3	0
Wasco	0	0	8	0
Washington	0	0	0	104
Wheeler	0	0	1	0
Yamhill	0	0	0	17

The result shows the relationship between Oregon's three CSA's and 36 counties (counties that are not in a specific CSA have coded as the CSA 'None' in our data). For example, CSA Bend consists of only two counties, whereas the Portland CSA is a collection of five counties. The resulting table object can be manipulated using the same subsetting commands used with matrices; the column and row names, similarly, are accessible via `rownames` and `colnames`.

<sup>3</sup>CSAs represent groupings of counties with close social and economic ties; we have coded the typically verbose CSA names with the name of the largest city contained within each area.

### 3.3 Histogram

Tables provide a **quick summary of categorical variables**. In some cases a numeric variable may have a small discrete set of potential values, in which case a table can also be useful for summarizing the data. For example, a variable giving the number of bedrooms in a dataset of housing units must be a nonnegative integer, typically between 0 and 5. Many numeric variables are either continuous (where nearly every value is unique) or take far too many unique values to be represented well by a table.

A **histogram** is a visualization used to quickly understand the distribution of a **numeric variable**. The R function `hist` produces this visualization from a numeric vector; the resulting graphic may be displayed in a new window, tab, frame, or application depending on the installation being used.<sup>4</sup> For example, the following code produces a histogram of the people per household by census tract.

```
> ppPerHH <- geodf$population / geodf$households
> hist(ppPerHH)
```

The result is shown in Fig. 3.1a. The height of each bar shows the number of data points that fall between its ranges. We see that most tracts have between 2 and 3 people per household, with a smaller set in the ranges 1–2 and 3–4. Looking closely at the small bump all the way to the right, there is at least one tract with as many as 9 people per household. **A default labeling of the plot and its axes has been given.**

Much of the visual space in the default histogram is taken by representing a few outliers on the right half of the window. We can remove these by filtering out any tract with more than five people per household.

```
> hist(ppPerHH[ppPerHH < 5])
```

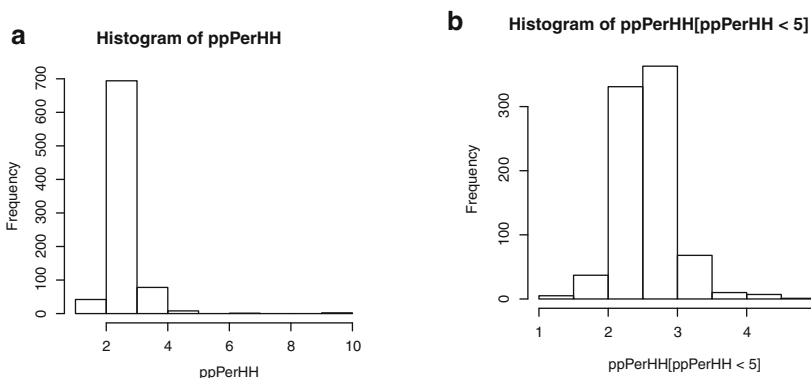


Figure 3.1: (a) Default histogram of population by household at a census tract level. (b) Histogram of population by household at a census tract level, truncating households of five people or more.

<sup>4</sup>We discuss how to save these as external images in Sect. 5.2.

The new histogram is shown in Fig. 3.1b, with the scale now ranging from 1 to 5. The bars have automatically adjusted so that each bucket has length 0.5, compared to the previous plot that had buckets of length 1. The finer grain shows that the tracts with between 2 and 3 people are approximately evenly distributed between 2 and 2.5 and 2.5 and 3, with only slightly more in the latter group.

By default, the histogram command automatically tries to pick “nice” intervals for breaking up the data. We have already seen that the length of a bucket decreases as the range decreases, so that the number of bars remains relatively constant. Also, the function attempts to pick interpretable break points; in our first example these are whole integers, and in the second these are half integers. The exact algorithm comes from a classic paper by [Herbert Sturges published in 1926](#) [2].

As dataset sizes have grown considerably over the past 90 years, it is often desirable to increase the number of buckets to better understand the fine grain detail of the data. The buckets can be manipulated by passing an input to the parameter `breaks`. A single integer input to `breaks` indicates that the plotting function should construct `breaks` number of (equally sized) intervals. For example, the following asks for 30 buckets.

```
> hist(ppPerHH[ppPerHH < 5], breaks=30)
```

The output, Fig. 3.2a, shows a finer grain plot than our previous version. We now see that the distribution has an approximately bell-shaped curve, centered somewhere around 2.6 people per household. If we count the buckets, we see that the `hist` function has taken our request for 30 intervals as a suggestion; it ultimately decided to split at every tenth of a unit resulting in 34 buckets.<sup>5</sup> The histogram command, even when asking for a fixed number of breaks, attempts to find “nice” split points and uses them instead of exactly producing the desired bucket count. Usually this behavior is helpful. We can change the coarseness of the plot as needed while allowing R to construct interpretable break points. Occasionally, it is necessary to manually set the split points of the histogram. To do this, the `breaks` parameter takes a vector of split locations. These are taken verbatim and will not be manipulated by the plotting function. For example, to recreate the breaks from Fig. 3.2a, we can specify the following:

```
> hist(ppPerHH[ppPerHH < 5], breaks=(13:47) / 10)
```

The code `(13:47) / 10` is simply a convenient shortcut for specifying a sequence of between 1.3 (where the x-coordinate starts) and 4.7 (where the y-coordinate ends) in steps of size 0.1. Manually setting histogram breaks can be useful when creating a collection of plots where it can be helpful to keep the scales consistent. When the numeric data are from a discrete set of numbers, such as grades on a 0–100 scale, it may also be helpful to construct a histogram where each bucket has exactly one unique value inside of it.

---

<sup>5</sup>We still count buckets even if they are zero. So, in our case, there are three empty buckets.

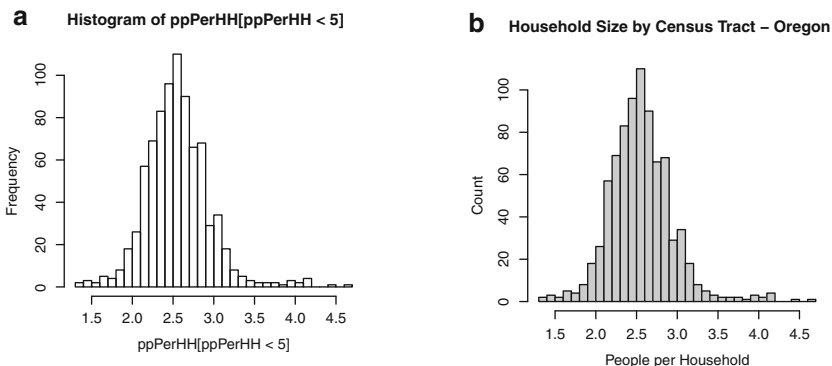


Figure 3.2: (a) Histogram of population by household at a census tract level, setting the number of break points to 30 (which is used only as a suggestion). (b) Histogram of population by household at a census tract level, with title, axis labels, and shading.

The `hist` function allows for a great deal of aesthetic customization. For example, the parameter `main` can be used to set the title of the plot. Similarly, `xlab` and `ylab` provide means for altering the axes labels. Setting the color parameters shades the inside of the buckets with the desired color. For a list of available color names type `colors()` in the R terminal; most of the standard English color names are available, along with hundreds of additional shades.

```
hist(ppPerHH[ppPerHH < 5], breaks=30,
     col="gray",
     xlab="People per Household",
     ylab="Count",
     main="Household Size by Census Tract - Oregon")
```

The result in Fig. 3.2b shows the same core histogram, but with buckets shaded in grey and the descriptive custom plot labels.

## 3.4 Quantiles

We now load a separate dataset derived from the ACS describing primary means of transportation to work. The rows correspond exactly to the original dataset `geodf`. As the entire dataset consists of numeric values, we will immediately convert the result of reading into R as a matrix object.<sup>6</sup>

```
> meansOfCommute <- read.csv("data/ch03/meansOfCommute.csv",
+                             as.is=TRUE)
> meansOfCommute <- as.matrix(meansOfCommute)
```

<sup>6</sup>Recall that `read.csv` will always return a data frame.



```
> meansOfCommute[1:5,]
      total car public_trans bus subway railroad ferry bike walk taxi
[1,]  1242  984           0  0      0           0  0    8  62  40
[2,]  1199 1036           0  0      0           0  0   34  38  14
[3,]   999  849           0  0      0           0  0   21  16   0
[4,]  1035  954           0  0      0           0  0    4  43   0
[5,]  1123  938           0  0      0           0  0   40  55  21

      work_at_home
[1,]           148
[2,]            77
[3,]           113
[4,]            34
[5,]            69
```

The first column gives a total population count, which may differ slightly from the population in the original dataset for various technical reasons.<sup>7</sup> Other columns count the primary modes of transportation to work. The first few rows indicate that these are highly skewed, with more common modes of transit such as “car” greatly preferred to more specialized forms such as “ferry”.

Raw counts of this data are not particularly interesting as the census tracts are not all the same size. Working with percentages makes an easier comparison between rows of data. Let us say we want to know the percentage of the population walking to work. Notice that we can access a column of the matrix by using the column name as an index inside the square braces.<sup>8</sup>

```
> walkPerc <- meansOfCommute[, "walk"] / meansOfCommute[, "total"]
> walkPerc <- round(walkPerc * 100)
```

We round the percentage of walkers to a whole number to make it easier to work with the output.

To explore the distribution of this variable, we might start by constructing a histogram as in the previous section. A popular alternative is the *five number summary*, which is returned by default by the `quantile` function.<sup>9</sup>

```
> quantile(walkPerc)
 0%  25%  50%  75% 100%
  0    1    3    5   50
```

The first and last numbers give the minimum and maximum value of the variable, respectively. Accordingly, there is at least one tract where no one walks to work and one tract where half of the people walk to work. The middle number is the *median* of the dataset, defined as the value such that half of the input values are less than the

<sup>7</sup>The Census Bureau will not release counts that might identify an individual. For example, if only one person takes a ferry to work from a given tract, it will not be included in either the total or ferry column. As a result, population numbers can vary slightly across the data set.

<sup>8</sup>This is the fourth and final method for subsetting as referenced in Sect. 2.5.

<sup>9</sup>There is, mostly for historical reasons, also a function `fivenum` in base R that returns the same numbers.

median and half of the input values are greater than it.<sup>10</sup> The second number is the 1st (or lower) quartile, often represented by the symbol  $Q_1$ ; it is defined similarly to the median, such that  $1/4$  of the input values are less than it and  $3/4$  of the input values are greater than it. The fourth number is the 3rd (or upper) quartile and is defined analogously; in other words, this means that in 75 % of the tracts, 5 % or less of the population walks to work.

The five number summary gives a compact description of a numeric variable. In this case we know that at least one tract has zero walking commuters, whereas at least one has one out of every two people commuting by foot. A typical census tract has roughly 1–5 % of the population commute by walking, with 3 % being the most typical value. The median being closer to the first quartile compared to the third quartile, and much closer to the minimum compared to the maximum, indicates that the data are *positive skewed*. In other words, there are a sizable number of oddly large values compared to the oddly small values. Recomputing the five-number summary for the percentage of people commuting by car gives an example of a dataset that is *negative skewed*.

```
> carPerc <- meansOfCommute[, "car"] / meansOfCommute[, "total"]
> carPerc <- round(carPerc * 100)
> quantile(carPerc)
 0%  25%  50%  75% 100%
22   78   85   89   98
```

Here there are a sizable number of particularly small values compared to overly large values.

The five number summary is an example of the more general concept of *quantiles*. If a numeric dataset is sorted and divided up into  $q$  equally sized buckets, the  $q$ -quantiles give the breakpoints for these breaks. The five number summary consists of the 4-quantiles, also known as quartiles (hence the name of the 1st and 3rd). Another common set of quantiles are the 10-quantiles, or deciles. To calculate these with the `quantile` function we pass the set of desired probabilities to the `prob` parameter.

```
> (0:10)/10
[1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
> quantile(walkPerc, prob=(0:10)/10)
 0%  10%  20%  30%  40%  50%  60%  70%  80%  90% 100%
 0    0    1    2    2    3    3    5    6   10   50
```

By setting the `prob` parameter to a range of values between 0 and 1 in steps of size 0.1, the deciles are returned. These give additional information to the distribution of the walking as a means of commuting, particularly with the higher deciles. We see that tracts with more than 10 % of the population walking to work are particularly rare, indicating that the one value of 50 % is particularly anomalous.

By far the most frequently used quantile in nontechnical work is the 100-quantile, also known as a percentile. In order to help calculate these, we use the function `seq` which constructs a sequence of numbers between a start point and

<sup>10</sup>Technically, if there are  $n$  data points, for odd datasets it is the  $\frac{n+1}{2}$ th largest value and for even datasets it is the average of the  $\frac{n}{2}$ th and  $(\frac{n}{2} + 1)$ th largest variables.



end point using a given step size.<sup>11</sup> To simplify the output, we also set the `names` parameter in the `quantile` function so that only the values and not their names are returned. Additionally, we will round with the `round` function to three decimal places.

```
> cent <- quantile(walkPerc,prob=seq(0,1,length.out=100),
+                 names=FALSE)
> round(cent,1)
[1] 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
[14] 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
[27] 1.0 1.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0
[40] 2.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0 3.0 3.0 3.0 3.0 3.0
[53] 3.0 3.0 3.0 3.0 3.0 3.0 3.0 3.0 3.0 3.0 4.0 4.0 4.0
[66] 4.0 4.0 4.0 4.0 4.0 5.0 5.0 5.0 5.0 5.0 5.0 5.0 6.0
[79] 6.0 6.0 6.0 7.0 7.0 7.0 7.0 8.0 8.0 9.0 9.0 10.0 10.0
[92] 11.0 12.0 13.0 15.0 18.0 20.0 22.3 31.7 50.0
```

The middle values are somewhat hard to parse, but the percentiles do a nice job of characterizing the extreme ends of the distribution. For instance, we see that about 10 % of the data have almost no walking commuters, whereas 4 % (i.e., the top 4 centiles) have at least one in five people walking to work.

For understanding the entire distribution of a numeric variable, histograms often give an easier to digest representation of the data's distribution. Quantiles can be more useful when looking at the extreme values of skewed data. The five number summary is also very useful in communicating the results of an analysis because a single five column matrix can summarize datasets with dozens of variables.

Another powerful application of quantiles is their ability to be used in subsequent analyses. For example, the quantile at 10 % can be used to label tracts with particularly low car usage.<sup>12</sup> The resulting categorical variable can be used in conjunction with the CSA designations, which were loaded in Sect. 3.2, to understand whether there is any regional trend to the presence of low-car usage.

```
> coff <- quantile(carPerc, prob=0.10)
> coff
10%
66
> lowCarUsageFlag <- (carPerc < coff)
> table(lowCarUsageFlag, geodf$csa)

lowCarUsageFlag Bend Medford None Portland
FALSE          28          54      336      327
TRUE           0           3       23       55
```

By looking at those tracts where car usage is among the lowest decile (specifically, 66 % or fewer people drive to work), we see that the majority of the low-car using tracts are located in the Portland metro area. This result is not too surprising given the increased housing density and prevalence of public transit within Oregon's largest metropolitan area.

<sup>11</sup> It is also possible to specify the desired output length and let the function determine the necessary step size.

<sup>12</sup> The R function can handle any vector of **probabilities in the `prob` parameter** not just formal  $q$ -quantiles. We make extensive use of this throughout this text.

## 3.5 Binning

There is a close conceptual link between histograms and quantiles. Histograms take evenly size intervals and count the amount of data within each interval, whereas quantiles take equal proportions of a numeric dataset and calculate where the splits occur. Histograms are also typically displayed as a graphic and quantiles as a string of numbers, but both can exist in either tabular or graphical form.<sup>13</sup> *Binning* is a conceptual cousin to both of these methods and is a generalization of the previous example where car usage percentages were bifurcated into two categorical variables. Data samples are placed into groups, the “bins”, according to where one variable falls in a set of cut-off values. Often these cut-off values come from a set of quantiles, though they may be any set which spans the range of the variable of interest.


Consider the deciles of the proportion of people commuting by car by census tract.

```
> breakPoints <- quantile(carPerc, prob=seq(0,1,length.out=11),
+                           names=FALSE)
> breakPoints
[1] 22 66 76 80 83 85 87 89 90 92 98
```

An interesting next step is to assign each data point to one of the ten buckets implied by the deciles. This can be done by the `cut` function available in R. It takes a numeric vector and set of break points and assigns each point to one of the implied buckets. Two parameters that should be altered from their defaults are `labels` and `include.lowest`. Setting these to `FALSE` and `TRUE` yield the most useful results for our purpose.<sup>14</sup>

```
> bin <- cut(carPerc, breakPoints, labels=FALSE, include.lowest=TRUE)
> bin[1:42]
[1] 3 6 5 9 5 3 2 4 4 2 6 2 1 1 1 1 7 5 5 4 1
[22] 1 3 3 4 3 3 4 5 6 4 2 7 6 5 4 6 3 6 3 7 5
> table(bin)
bin
 1    2    3    4    5    6    7    8    9   10
87  93  86  89  68  94 104  45  88  72
```

The values returned by `cut` are integer values assigning each input into one of the deciles. For example, the fourth row of the dataset is in the ninth decile (the second highest) of car commuters. A table of the bins shows that the number of samples in each is similar but not completely uniform. As we used quantiles for the cut offs, it would be reasonable to assume that the sizes would all be the same. The discrepancy comes about because we rounded the percentages. Looking at the raw data, we see 51 tracts with 89% car commuters, which are all currently placed in bin 7. If these were moved to bin 8 (the smallest), bin 7 would be left with only 53

<sup>13</sup>A visualization of two sets of quantiles, a *Q-Q plot*  very popular statistical tool. It is used mostly in conjunction with inferential statistics, which is outside the scope of this text.

<sup>14</sup>We encourage readers to try `cut` with other choices to see why these are needed.

tracts. In short, there is no way to evenly split the tracts up given the discreteness of the data; the `quantile` and `cut` functions do the best job possible.

We will see many uses for the values from binning data. Typically these consist of using techniques designed for categorical variables with numeric ones by transforming into bin ids. For example, using methods already covered, we can now use the bin ids to look at our ten bins in a two-way table showing the relationship between car ownership and CSA designation.

```
> table(bin, geodf$csa)
```

bin	Bend	Medford	None	Portland
1	0		3	24
2	2		5	28
3	4		4	30
4	1		3	41
5	1		3	34
6	1		5	44
7	8		15	43
8	2		5	22
9	6		8	48
10	3		6	45

We see again that Portland dominates the low end of the distribution; those tracts not in a CSA (likely to be the most rural) dominate the upper quantiles. Given that in rural areas automobiles are often the only method of transit, this result confirms our common assumptions about car usage.

While it is often useful to bin data based on the result of calling the `quantile` function, the `cut` function may be used to create buckets with any set of break points.<sup>15</sup> One option is to use equally spaced cuts for the binning. For example, consider cutting the people per household data (truncated from above at 5) using a sequence of breaks based on what we did previously in Sect. 3.3.

```
> bins <- cut(ppPerHH[ppPerHH < 5], breaks=seq(1.3,4.7,by=0.1),
+           labels=FALSE, include.lowest=TRUE)
> table(bins)
```

bins																
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
2	3	2	5	4	8	18	26	57	69	83	96	110	90	66	68	29
18	19	20	21	22	23	24	25	26	27	28	29	32	34			
34	18	8	5	3	2	2	2	1	3	2	4	1	1			

The resulting bins, seen as a table, are the exact heights of the bars shown in the histogram from Fig. 3.2a! Likewise, if we used quantiles as the breaks in a histogram, the resulting histogram will consist of bins with equal area, rather than equal widths. This is shown in Fig. 3.3.<sup>16</sup>

<sup>15</sup>If the break points do not span the data set, missing values will be returned for data outside the range of breaks.

<sup>16</sup>We originally described the height of the bins in a histogram as the counts, but this was only because the widths of the buckets were the same. In reality, it is the area of the buckets that matter; these are not the same with unequally spaced cuts.

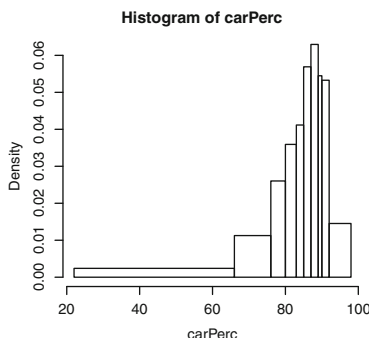


Figure 3.3: Histogram of the percentage of census tract residents who primarily use a personal automobile to commute to work. Uneven breaks are used, with splits occurring at the deciles of the data. Each histogram bar, therefore, has approximately the same area.


```
> hist(carPerc, breaks=breakPoints)
```

Clearly these three techniques are closely related. Successful data exploration of a single continuous variable requires adeptly employing all three to their respective strengths.

### 3.6 Control Flow

The next set of data from the ACS we will look at gives counts of household-level income. As before, we read the csv file into R and convert the result into a numeric matrix, which has the same number of rows as the data frame `dfgeo`. We do need one additional parameter to `read.csv` called `check.names` to be manually set to `FALSE` to prevent R from changing the column names.<sup>17</sup>

```
> hhIncome <- read.csv("data/ch03/hhIncome.csv", as.is=TRUE,
+                      check.names=FALSE)
> hhIncome <- as.matrix(hhIncome)
> hhIncome[1:5,]
  total 0k 10k 15k 20k 25k 30k 35k 40k 45k 50k 60k 75k 100k 125k
1  1225 113  60  42  87  50  27  48  90  77  86 183  94  132  86
2  1322 119 162  93  81  91  15  27  72  43 139 113 160   56 106
3  1162 107  69 107 116  60 127  44  88  92 129  86  94    4    8
4  1397 168 188  89  98  84  89  51  58  83 107 146 140   34   24
5  1117  70 131  93  73  66  60  76  89 112 107  90  93   29    3
```


<sup>17</sup>The column names we constructed start with a number, which is technically allowed for column names but not allowed for object names.  means that, as a data frame, we would not be able to access the columns of `hhIncome` using `hh` operator. As we are converting into a matrix this is not a concern, so we instruct R not to try to paste an “X” to the front of all the variable names using `check.names=FALSE`.

	150k	200k
1	24	26
2	22	23
3	5	26
4	24	14
5	0	25

We have labeled the columns with short names to make them easier to access and print. Column “20k” represents a count of the number of households that earned between 20,000 and 25,000 (the next column name) dollars per year. The final column denotes the number of households that earns 200,000 or more dollars per year. The first column represents the total number of households represented by this analysis.

Consider just one row of data from this set, removing the first column of totals.

```
> oneRow <- hhIncome[1, -1]
> oneRow
 0k  10k  15k  20k  25k  30k  35k  40k  45k  50k  60k  75k 100k 125k
113   60   42   87   50   27   48   90   77   86  183   94  132   86
150k 200k
 24   26
```

The raw data are interesting in their own right, but perhaps more insightful would be to convert this into a cumulative count of the number of households that have an income below some threshold. We could calculate  of these cumulative values by hand, but R provides a function called `cumsum`. The function takes a numeric input and cumulatively adds each element to the one before it. For our row of data the result is as follows:

```
> cumsum(oneRow)
 0k  10k  15k  20k  25k  30k  35k  40k  45k  50k  60k  75k 100k 125k
113  173  215  302  352  379  427  517  594  680  863  957 1089 1175
150k 200k
1199 1225
```

So 863 of the households in this tract make less than 75k dollars per year. Notice that we need to look at the number right before the value labeled 75k (the one for 60k) rather than its value directly.

How might we go about computing this cumulative sum for the entire dataset? Clearly we would not do this manually for each row of data; at worst we would forgo the `cumsum` function and manually compute each column using vectorized notation (there are only 16 columns but hundreds of rows). Fortunately, we do not have to resort to either method. The R language provides a set of tools for what is referred to as *control-flow constructs*. One example of these is a *for loop*, which iteratively executes a block of code, with one variable taking on each and every value in a given vector. That is a mouthful that can be better explained through an example. Take the set of integers `1:5`. The *for loop* in the following code

```
> for (j in 1:5) {
```

```
+ print(j)
+ }
```

will actually execute the following:

```
> j <- 1
> print(j)
> j <- 2
> print(j)
> j <- 3
> print(j)
> j <- 4
> print(j)
> j <- 5
> print(j)
```

So when we run the *for loop* in R, this is what we see:

```
> for (j in 1:5) {
+   print(j)
+ }
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

The function `print` needs to be called explicitly, as inside a *for loop* printing of unassigned variables is not done automatically as would otherwise be the case. In all other respects, the code executed by the *for loop* is exactly the same as the expanded version given.

With this new construct, we can now use a *for loop* to cycle over the rows of our dataset. As is often the case with such loops, prior to doing that we need to construct an empty matrix in which we will store the results. Here, we make a matrix filled with zeros that contain the same number of rows and one fewer column (as we do not need one for the total) than the raw data.

```
> cumIncome <- matrix(0, ncol=ncol(hhIncome)-1, nrow=nrow(hhIncome))
```

We now write a *for loop* where the index `j` cycles over a sequence of numbers from 1 to the number of columns in `hhIncome`. In each iteration, the cumulative sum function is applied to a given row and the counts are divided by the total of the row. The calculation is saved in a row of our result matrix `cumIncome`.

```
> for (j in 1:nrow(hhIncome)) {
+   cumIncome[j,] <- cumsum(hhIncome[j,-1]) / hhIncome[j,1]
+   cumIncome[j,] <- round(cumIncome[j,] * 100)
+ }
> colnames(cumIncome) <- colnames(hhIncome)[-1]
```

At the end, we assign the correct column names to the outcome vector.

We see that the desired values have been filled into our matrix.

```
> cumIncome[1:5,]
      0k 10k 15k 20k 25k 30k 35k 40k 45k 50k 60k 75k 100k 125k 150k
[1,]   9  14  18  25  29  31  35  42  48  56  70  78   89   96   98
[2,]   9  21  28  34  41  42  44  50  53  64  72  84   89   97   98
[3,]   9  15  24  34  40  50  54  62  70  81  88  96   97   97   98
[4,]  12  25  32  39  45  51  55  59  65  73  83  93   96   97   99
[5,]   6  18  26  33  39  44  51  59  69  79  87  95   97   98   98

      200k
[1,]  100
[2,]  100
[3,]  100
[4,]  100
[5,]  100
```

As a nice check, the entire final column is filled with 100s, which is expected as 100% of the data for each row should be included when cumulatively summing all of the columns. We will use this cumulative data later in Chap. 4.

### 3.7 Combining Plots

Returning to the original household income data, notice that it has 16 numeric columns. If we were actually fully exploring this data, a first good step would be to pick a single column and to build exploratory histograms and quantile bins to first understand it by itself.<sup>18</sup> As a second step, it would be nice to automate this process so that we do not have to manually construct 16 sets of exploratory data figures. The *for loop* makes this relatively easy.

R provides a way of adding multiple plots to the same graphics window using the function `par`, which sets the parameters of our graph. Prior to any plots being drawn, the `par` function is called with the parameter `mfrow` equal to a vector of two integers; subsequent graphics will be plotted *by row* in a grid with dimensions equal to the inputs to the `mfrow` parameter.<sup>19</sup> Drawing all 16 histograms is then possible through the following.<sup>20</sup>

```
> par(mfrow=c(4,4))
> for(j in 1:16) {
+   hist(hhIncome[,j+1] / hhIncome[,1],
+       breaks=seq(0,0.7,by=0.05), ylim=c(0,600))
+ }
```

To begin adjusting the plots aesthetics, we specified the exact break points and limits on the *y*-axis so that each of these histograms is on exactly the same scale.

<sup>18</sup>In the interest of verbosity, we skip that here but encourage readers working along to try this themselves.

<sup>19</sup>A similar parameter `mfcol` can be used to plot figures by column.

<sup>20</sup>It is possible that you will get the error `Error in plot.new() : figure margins too large`, which can often be remedied by closing the plot window and running the code from scratch.

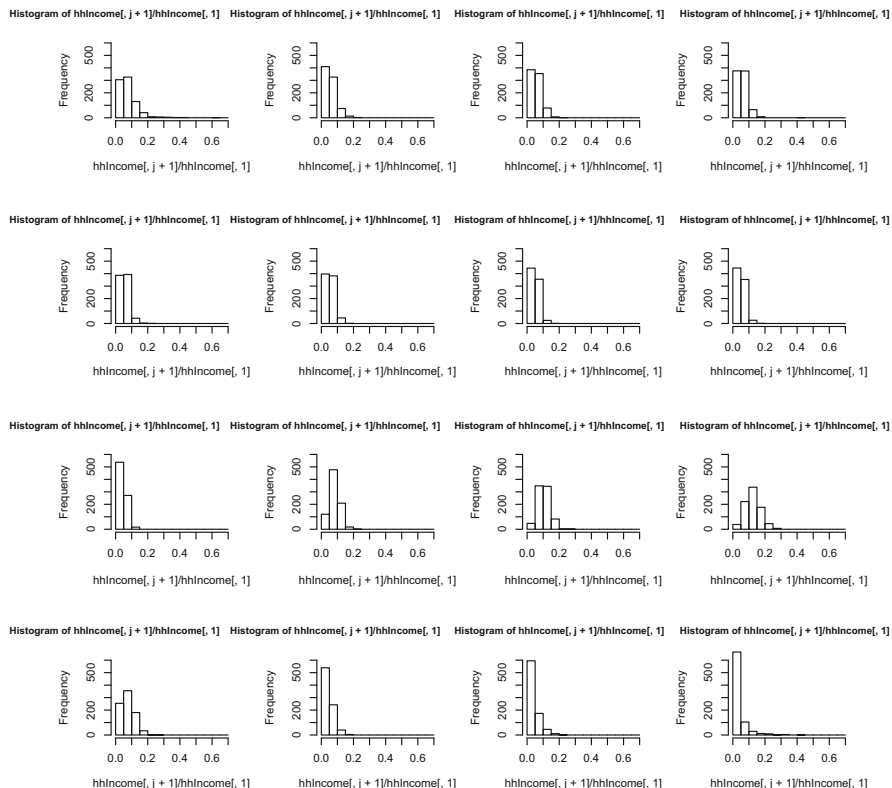


Figure 3.4: Grid of income band distributions at a census tract level using default margins and titles. Axes ranges are held constant.

The output is shown in Fig. 3.4. As we may have expected, the distribution of proportions becomes higher for the middle ranges before shrinking towards zero again for the higher income bands. In other words, there are typically the highest counts of households making 50–100k dollars per year, compared to the counts of households making significantly less or significantly more.

The default histograms plotted in our grid does a reasonable job of conveying the distribution of income throughout the tracts in the state of Oregon. However, it is not possible to read the income bands off of the plot, and a lot of space is wasted on uninteresting default labels. How might we fix this? To start, we need a vector of length 16 giving a nice label for each plot. This can be constructed by pasting together the column names from our data set. After removing the first name “total”, the first 15 names are pasted, with a dash in between them, to the last 15 names to show the ranges given in the buckets. The final bucket label is manually added.

```
> bands <- colnames(hhIncome)[-1]
> bandNames <- paste(bands[-length(bands)], "-", bands[-1], sep=" ")
> bandNames <- c(bandNames, "200k+")
```



```
> bandNames
[1] "0k-10k"      "10k-15k"      "15k-20k"      "20k-25k"      "25k-30k"
[6] "30k-35k"      "35k-40k"      "40k-45k"      "45k-50k"      "50k-60k"
[11] "60k-75k"      "75k-100k"     "100k-125k"    "125k-150k"    "150k-200k"
[16] "200k+"
```

These can now be plotted over the histograms using the function `text` that puts a label at a given set of coordinates.<sup>21</sup> As all of the plots have the same  $x$  and  $y$  ranges, we can hard code this to be at  $x = 0.33$  and  $y = 500$ .

In order to make the plot fill up more of the image with interesting data rather than repeating titles and axes labels, three steps are needed. Within the `hist` function call, we set `axes` to `FALSE`, and the labels `xlab`, `ylab`, and `main` to empty strings. Secondly, we make another call to the `par` function, setting the parameter `mar` (margin) to a vector of four zeros. These values correspond to the bottom, left, top, and right margins of the plot; we are setting them all to zero. Finally, because turning off the axes also turns off the box around the plot, which is visually useful in a grid of histograms, we make an additional call to the function `box` in each iteration.

```
> par(mfrow=c(4,4))
> par(mar=c(0,0,0,0))
> for(j in 1:16) {
+   hist(cumIncome[,j], breaks=seq(0,1,length.out=20), axes=FALSE,
+       main="", xlab="", ylab="", ylim=c(0,600), col="grey")
+   box()
+   text(x=0.33,y=500,
+       label=paste("Income band:", bandNames[j]))
+ }
```

The output from this code is shown in Fig. 3.5. It is much more visually pleasing and conveys additional meaning from the original plot due to the inclusion of readable band labels.

### 3.8 Aggregation

Another use of *for loops* is to aggregate a continuous variable over the unique values of a categorical one, for example, summing up the tract populations in `geodf` by their CSA designation to get a population total for each of the combined statistical areas in Oregon. As before, we first need to create an empty object in which to store the output of the *for loop*. Here we do this with the help of the `unique` function, which returns the unique values of a vector, in order to construct a named vector of population totals.

```
> csaSet <- unique(geodf$csa)
> popTotal <- rep(0, length(csaSet))
> names(popTotal) <- csaSet
```

<sup>21</sup>We will see more advanced uses of this function in Sect. 4.

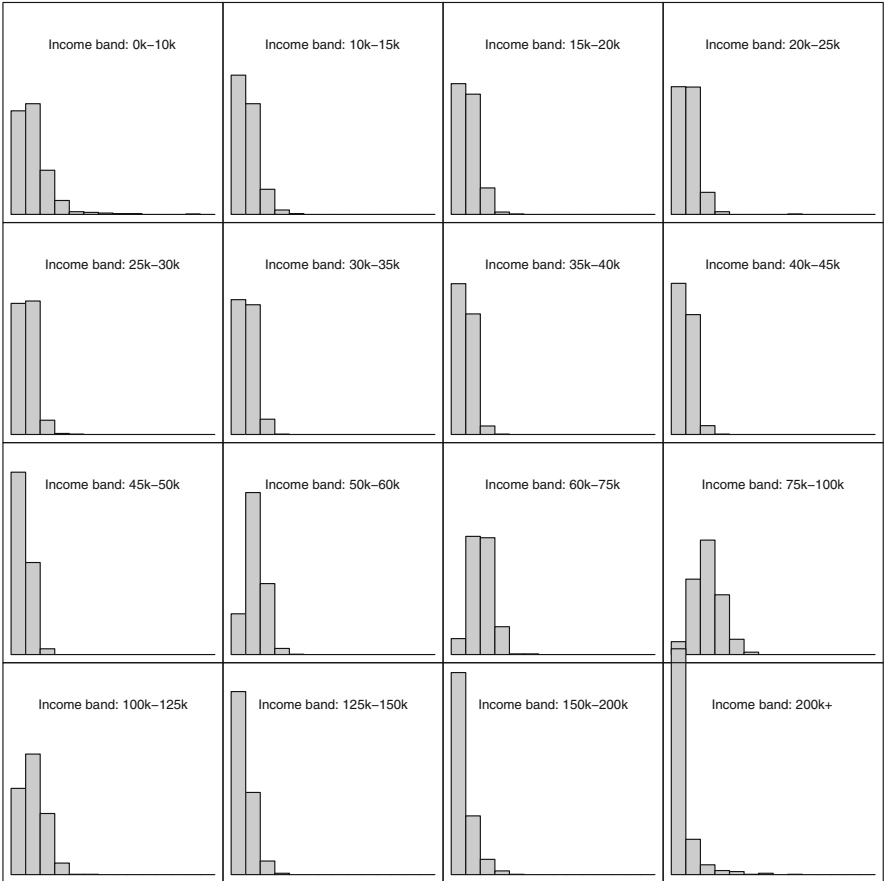


Figure 3.5: Grid of income band distributions at a census tract level with repressed margins and titles. Axes ranges are held constant across the plots. Income ranges are printed in each box.

The data frame is then looped over by row, with each row assigned to a particular element of `csaSet` using the `match` function and its population added to the corresponding running total.<sup>22</sup>

```
> for (j in 1:nrow(geodf)) {
+   index <- match(geodf$csa[j], csaSet)
+   popTotal[index] <- popTotal[index] + geodf$population[j]
+ }
> popTotal
      None Portland      Bend  Medford
1582446 1816916 181459 287900
```

<sup>22</sup>See Sect. 2.9 for a refresh of the `match` function, a common point of confusion for new R users.

The resulting vector shows that 1.8 million people live in the Portland CSA (at least within Oregon; technically the region spills over the state boundary into Washington State).

Looping over the rows of the data frame is not the only method of using loops to aggregate over the four CSA designations. An alternative is to cycle over the set of CSA values, extracting the matching rows of the data frame and adding them together in each iteration. As an example, here is an example implementing this aggregation to count the number of individuals who work at home in each CSA region.

```
> csaSet <- unique(geodf$csa)
> wahTotal <- rep(0, length(csaSet))
> names(wahTotal) <- csaSet
> for (csa in csaSet) {
+   index <- which(geodf$csa == csa)
+   wahTotal[csa] <- wahTotal[csa] +
+     sum(meansOfCommute[index,"work_at_home"]) * 100
+ }
```

Dividing this count by the total population shows the percentage of each CSA who works from home.

```
> wahTotal / popTotal
      None  Portland      Bend  Medford
2.488805  3.083247  3.177026  2.755818
```

It is commonly known that R slows down considerably when executing loops; so this second method will scale better when working with large data sets as it requires fewer iterations. At the scale of data we are working with here, though, the difference is practically unnoticeable.

### 3.9 Applying Functions

We have used loops to apply a function over the rows of a matrix. In addition to this method, R has a special syntax for applying functions over the rows or columns of a matrix using the `apply` function. The following code snippet, for example, applies the function `sum` over the rows of the matrix `meansOfCommute`. The parameter `MARGIN` determines which dimension to apply the function over, with rows having a value of 1 and columns having a value of 2.

```
> apply(meansOfCommute[1:10,-1],MARGIN=1,FUN=sum)
[1] 1242 1199  999 1035 1123  729 4080 2042 3608 1315
> meansOfCommute[1:10,1]
[1] 1242 1199  999 1035 1123  719 3932 2042 3559 1275
```

Looking at the `apply` function of the first ten rows, notice that the function has verified that the first column is the total of the other columns. Accordingly, 1242

is the aggregate of the means of commute such as car and bike. Applying the `sum` function over `MARGIN=2` shows the total counts of transportation types over the entire dataset.<sup>23</sup>

```
> apply(meansOfCommute,2,sum)
      total      car public_trans      bus      subway
1700451    1394475      70714    55784      5119
railroad      ferry      bike      walk      taxi
  2751         32     39789    69285     17085
work_at_home
  109103
```

We see, among other things, that only 32 people claim to take a ferry to work.

The `apply` function can be used even when the result of applying a function to a row is a vector rather than a single number. The result, assuming each vector is the same length, is a new matrix with the results combined together. As a practical example, consider the process of taking the cumulative sum of each row in the household income dataset. The following one line of code can replace the previous code block that used *for loops*:

```
> cumIncome <- apply(hhIncome[, -1], 1, cumsum)
> dim(cumIncome)
[1] 16 826
```

The result is a flipped version of our previous code because `apply` always combines the results by column. In order to exchange the rows and columns of a matrix, the function `t` (one letter, standing for *t*ranspose) is used.

```
> cumIncome <- t(cumIncome) / hhIncome[,1]
> cumIncome <- round(cumIncome * 100)
> cumIncome[1:5,]
      0k 10k 15k 20k 25k 30k 35k 40k 45k 50k 60k 75k 100k 125k 150k
[1,]   9  14  18  25  29  31  35  42  48  56  70  78   89   96   98
[2,]   9  21  28  34  41  42  44  50  53  64  72  84   89   97   98
[3,]   9  15  24  34  40  50  54  62  70  81  88  96   97   97   98
[4,]  12  25  32  39  45  51  55  59  65  73  83  93   96   97   99
[5,]   6  18  26  33  39  44  51  59  69  79  87  95   97   98   98

      200k
[1,]  100
[2,]  100
[3,]  100
[4,]  100
[5,]  100
```

The result being the exact same as constructed in Sect. 3.6.

Another variant of the `apply` function is the `tapply` function, which applies a function over one vector by the unique values of another vector. Among other things, this allows aggregations to be done in a single function call. The following, for example, applies the function `sum` to counts of people who work from home by the unique values of CSAs.

<sup>23</sup>The `MARGIN` and `FUN` parameters are typically unnamed and set positionally, as they are here.

```
> wahTotal <- tapply(X=meansOfCommute[, "work_at_home"],  
+                    INDEX=geodf$csa,  
+                    FUN=sum)  
> wahTotal  
      Bend  Medford      None Portland  
    5765    7934    39384    56020
```

The result is exactly the same as derived in Sect. 3.8, but with significantly less code and without the need to pre-calculate the unique values of the CSA variable.

Other variants of apply functions exist in R: `lapply`, `mapply`, `eapply`, `vapply`, `sapply`, and `parApply`. These are all well documented in their respective help pages; we will mostly stick to the two previously mentioned variants throughout this text as they provide the majority of the required functionality.

## References


- [1] UC Bureau. American community survey. *Washington, DC*, 2009, 2005.
- [2] Herbert A Sturges. The choice of a class interval. *Journal of the American Statistical Association*, 21 (153): 65–66, 1926.

## Chapter 4

### EDA II: Multivariate Analysis

**Abstract** In this chapter, techniques for exploring the relationship between multiple continuous variables are shown, using scatter plots as basic building blocks.


#### 4.1 Introduction

In Chap. 3 we presented numerous methods for exploring and visualizing datasets. These techniques were either restricted to a single variable, such as histograms and quantiles, or involved categorical variables. Here we investigate methods for exploring the relationship between two (or more) continuous variables. These visualizations allow for a high degree of customization by varying color, shape, sizes, and other graphical parameters. The power of using a programming language for creating graphs, rather than a point and click GUI, is shown 

#### 4.2 Scatter Plots

A scatter plot is a visualization of two numeric variables using a two-dimensional region. For each data point, a mark is placed at the horizontal location value of the first variable and the vertical position of the second variable. Drawing a basic scatter plot in R involves calling the `plot` function with the two vectors of interest. As an example, we plot the number of households and the population of each census tract:

```
> plot(geodf$households, geodf$population)
```



The output in Fig. 4.1a shows the default scatter plot, which roughly resembles similar plots frequently displayed in newspapers, magazines, television shows, and other popular media. An impressive amount of information is succinctly displayed in this plot. We can see the range of the two variables, visually identify outliers (a few tracts with almost no households, and a few with over 4000), and approximately

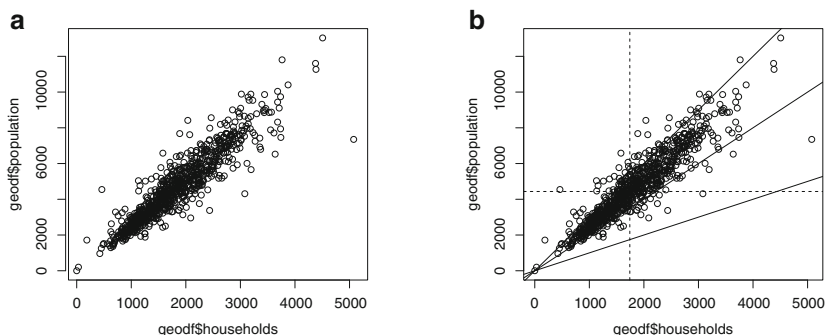


Figure 4.1: (a) Scatter plot showing number of households against population for census tracts in Oregon. (b) Scatter plot showing number of households against population for census tracts in Oregon. *Solid lines* plot have slopes equal to 1, 2, and 3 and run through the origin. The *dashed lines* denote the median of each component.

identify that most tracts have between 1000 and 2500 households and 2000–6000 people. The roughly linear relationship between the two variables is also seen, with most tracts having two to three people per household. The small set of outliers from this relationship are also quickly identifiable.

The flexibility of scatter plots in R comes mostly from the ability to layer additional information over a pre-existing plot. We have seen one example of this already when adding income band labels to histograms in Sect. 3.7. For the current scatter plot, a useful addition would be to add lines to help understand the ratio between population and the number of households. The function `abline` takes two numbers and draws a line using the first as the y-intercept (point where the line crosses the y-axis) and the second as the slope of the line. Running the following commands adds lines that run through the origin and have slopes of 1, 2, and 3. A point lying on the second line would, for example, represent a census tract with exactly two people per household.

```
> abline(0,1)
> abline(0,2)
> abline(0,3)
```

It is also possible to add vertical and horizontal lines using `abline`, by specifying (by name!) the parameter `v` or `h`. Here we add lines denoting the median of each coordinate through the parameter `prob=0.5`. The additional parameter `lty` indicates that these lines should be dashed rather than solid.

```
> abline(v=quantile(geodf$households,prob=0.5), lty="dashed")
> abline(h=quantile(geodf$population,prob=0.5), lty="dashed")
```

The output from these adding these lines is shown in Fig. 4.1b. While conveying the same underlying points, the new plot helps to strengthen and refine our previous observations. For example, we now see that the majority of tracts have between two and three people per household and none have fewer than one person per household.

We will now turn to another example and explore household income in Oregon looking specifically at income in the Portland metro area in relation to the rest of the state. In particular, we can further customize the outputs through additional parameters in our `plot` function to make our graph easier to read. We will use three of these functions frequently to manipulate the way the points are represented on the plot: `cex`, `pch`, and `col`. The parameter `cex` defines the relative size of the points (with one being the default); `pch` gives a number code to indicate the shape of the points; and `col` determines the color of the points. These can all be provided as single inputs, in which case they effect every point on the scatter plot. Otherwise, vectors of the same length as the input can be given with each element effecting only the corresponding data value. This provides a mechanism for seeing additional variables in the scatter plot, differentiated by color, shape, or size.

The values for `cex` are defined as multipliers to the default point sizes given the scale of the plot. Generally, we will not want a size smaller than 0.5 or larger than 2. Here we set the size of values corresponding to tracts outside of Portland to be 0.5, with those in the Portland metro area set to 1.

```
> cexVals <- rep(0.5, nrow(geodf))
> cexVals[geodf$csta == "Portland"] = 1
```

Making the points in Portland larger than the dots is helpful as otherwise the plus signs we are about to add will get lost in a sea of solid dots.

The two shape codes we use most are 19 (solid dots) and 3 (plus signs); for a complete list of the codes used in defining the input `pch`, see the help page by typing `?pch` in the R console. Here we will create vector `pch` values, first setting all of the values to 19 and then switching those corresponding to the Portland metro area to 3.

```
> pchVals <- rep(19, nrow(geodf))
> pchVals[geodf$csta == "Portland"] = 3
```

Using these values, points from tracts in the Portland area will be plus signs and other tracts will be solid dots.

Finally, to illustrate the point, we can define a custom vector of colors. We have already seen how to define color using character vectors as described in the list `colors()`. Another approach is through the function `grey`, which takes a number between 0 and 1 and returns shades of grey with lower numbers being darker and higher number lighter. For example, the following will make the Portland points a light grey and the non-Portland points a darker grey.

```
colVals <- rep(grey(0.2), nrow(geodf))
colVals[geodf$csta == "Portland"] <- grey(0.8)
```



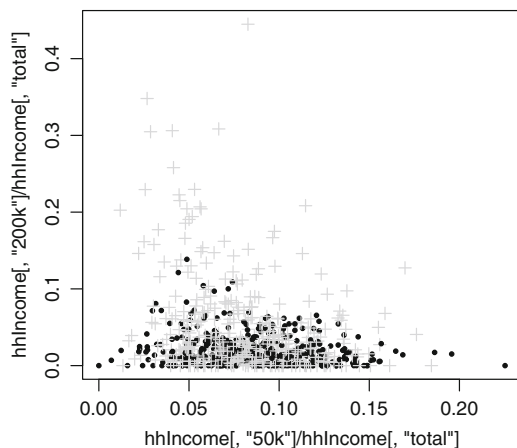


Figure 4.2: Scatter plot of tract level income band data from Oregon, showing the proportion of households earning \$45–50k per year against those earning \$75–100k. Tracts in the Portland metro area are denoted by *silver plus signs*; other tracts by *dark circles*.

The function `grey`, and its multi-chromatic version `rgb`, is useful when needing a larger set of colors as we can pass it a vector of values in order to get a long vector of shaded colors.

We now use these three graphical parameters in a scatter plot between the two income variables used in Sect. 3.6.

```
> plot(hhIncome[, "50k"]/hhIncome[, "total"],
+      hhIncome[, "200k"]/hhIncome[, "total"],
+      cex=cexVals,
+      pch=pchVals,
+      col=colVals)
```

The output shown in Fig. 4.2 shows the various differentiations between the Portland and non-Portland data points. We see that the set of tracts with a high percentage of households with incomes above \$200,000 are almost entirely from the Portland metro area.

### 4.3 Text

Using the `tapply` function introduced in Sect. 3.9, we can quickly calculate the proportion of households in each county that fall within a given income band.

```
> county30k <- tapply(hhIncome[, "30k"], geodf$county, sum)
> county200k <- tapply(hhIncome[, "200k"], geodf$county, sum)
> countyTotal <- tapply(hhIncome[, "total"], geodf$county, sum)
> county30k <- county30k / countyTotal
> county200k <- county200k / countyTotal
```

We could construct a scatter plot of the resulting data to understand the relationship between these two income buckets. Given that each data point has a well-defined name (the county) and the relatively small sample size, it would be even better if we could label these points with the county names.

In order to add labels to a plot, the `text` function is used. We already saw one application of this when adding income band labels to a grid of histograms. Here we give the function two vectors of coordinates and vector of labels in order to plot all the counties at once. In order to have the labels appear slightly above the points, rather than awkwardly plotting directly over them, we use an offset of 0.001 to the horizontal components.

```
> plot(county30k, county200k)
> text(county30k, county200k+0.001, labels=names(county30k), cex=0.5)
```

The `text` command accepts the same `cex` and `col`, which can again be a single number or a vector corresponding to each data point. The `pch` input is not applicable as text labels do not have a shape. Here we reduced the default sizes by half in order to keep the plot from becoming too cluttered.<sup>1</sup> The labeled plot is shown in Fig. 4.3. We see that there is not a clear linear relationship between these variables. Grant County has the second highest percentage of households making between \$30,000 and \$35,000 per year, and it is also one of the top counties for having top earners.

With the CSA values, we have a method for further grouping the individual counties. These groupings can be denoted on the plot using color. We first need to construct a vector with the same length of `county30k` and `county200k`, describing the corresponding CSA to each county. This is done via the `match` function:

```
> csaValues <- geodf$csa[match(names(county30k), geodf$county)]
> csaValues
 [1] "None"      "None"      "Portland"  "None"      "Portland"  "None"
 [7] "Bend"      "None"      "Bend"      "None"      "None"      "None"
[13] "None"      "None"      "Medford"   "None"      "Medford"   "None"
[19] "None"      "None"      "None"      "None"      "None"      "None"
[25] "None"      "Portland"  "None"      "None"      "None"      "None"
[31] "None"      "None"      "None"      "Portland"  "None"      "None"
```

These strings values now need to be converted into numeric levels; one way of doing this is to construct a set of the unique CSA values and save the raw index output of the `match` function.

```
> csaSet <- unique(geodf$csa)
> index <- match(csaValues, csaSet)
> index
 [1] 1 1 2 1 2 1 3 1 3 1 1 1 1 1 4 1 1 1 1 1 1 1 2 1 1 1 1 1 1
[33] 1 2 1 2
```

<sup>1</sup>Tweaks like this are generally done by trial and error.

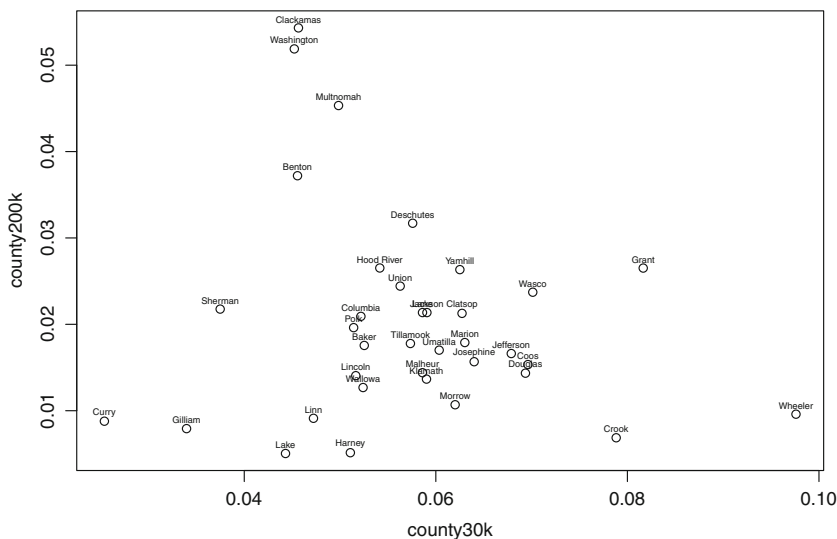


Figure 4.3: Scatter plot of county level income band data from Oregon, showing proportion of households earning \$25–30k against those earning \$200k or more.

Finally, we take a vector of four colors (one for each CSA) and construct a vector of color values matching the points in the scatter plot.

```
> colVals <- c("orchid1", "navy", "wheat3", "olivedrab")
> colVals[index]
 [1] "orchid1"  "orchid1"  "navy"      "orchid1"  "navy"
 [6] "orchid1"  "wheat3"   "orchid1"  "wheat3"   "orchid1"
[11] "orchid1"  "orchid1"  "orchid1"  "orchid1"  "olivedrab"
[16] "orchid1"  "olivedrab" "orchid1"  "orchid1"  "orchid1"
[21] "orchid1"  "orchid1"  "orchid1"  "orchid1"  "orchid1"
[26] "navy"     "orchid1"  "orchid1"  "orchid1"  "orchid1"
[31] "orchid1"  "orchid1"  "orchid1"  "navy"     "orchid1"
[36] "navy"
```

These colors are then used in both the plot and text functions.

```
> plot(county30k, county200k, col=colVals[index], pch=19)
> text(county30k, county200k+0.001, names(county30k),
+       col=colVals[index], cex=0.5)
```

Notice that the points have been set to solid dots using the `pch`; we will use this frequently in order to accentuate point colors.

The result of this labeled and colored plot is shown in Fig. 4.4. The mapping between colors and CSA values is given by the following data frame.

```
> data.frame(csaSet, colVals)
   csaSet  colVals
1   None  orchid1
```

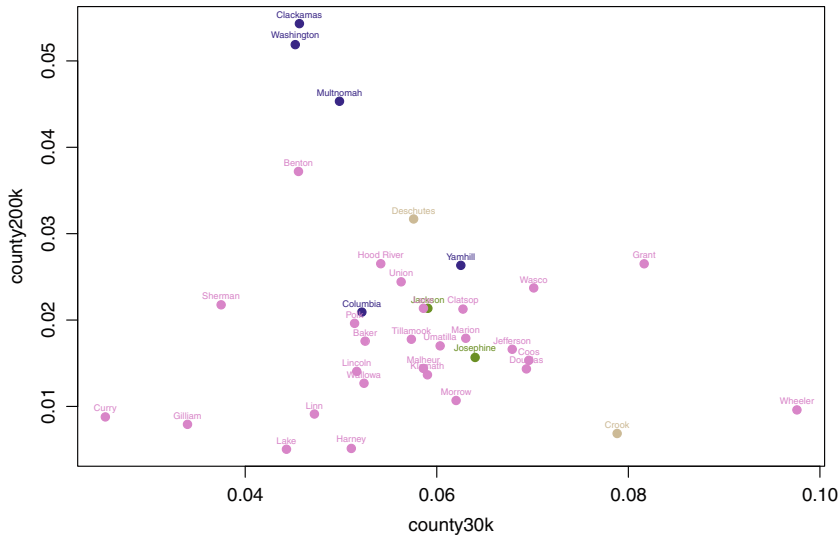


Figure 4.4: Scatter plot of county level income band data from Oregon, showing proportion of households earning \$25–30k against those earning \$200k or more. *Color* denotes the CSA of the county, with Portland counties in navy, Bend counties in *wheat* (dark yellow), and Medford counties in *olive*. Counties outside of a Combined Statistical Area are colored in *orchid* (purple-pink).

```
2 Portland      navy
3      Bend    wheat3
4      Medford olivedrab
```

For understanding the data, the plot does a good job of showing the relationship between the two income bands and labeling the points with the various county names and coloring based on the CSA of the county. The plot clearly has some aesthetic issues; some of the text labels run into one another, a legend mapping colors to CSAs should be shown directly on the graph, and the colors may not be ideal. We will address these and other concerns later in Chap. 5.

## 4.4 Points

Just as we might add text to an already constructed plot, it is also sometimes useful to add additional points to an existing scatter plot. Consider a scatter plot of two income band percentages, here at the basic tract level.

```
> plot(hhIncome[, "30k"] / hhIncome[, "total"],
+      hhIncome[, "200k"] / hhIncome[, "total"],
+      col="black",
+      pch=19,
+      cex=0.5)
```

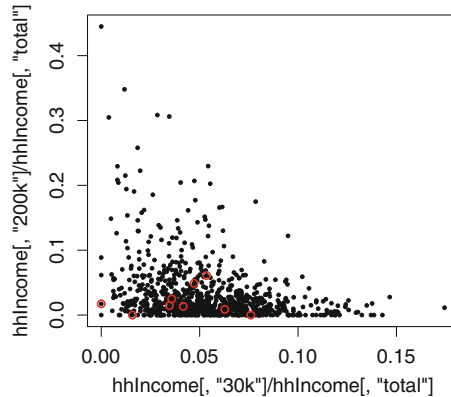


Figure 4.5: Scatter plot of tract level income band data from Oregon, showing the proportion of households earning \$25–30k per year against those earning \$200k+. *Red circles* highlight tracts where 30 % or more of the population walk to work.

If we are interested in studying places with a higher number of bike commuters, it would be helpful to call out where these points exist on the plot. One method would be to color them differently in the first place; alternatively we can identify them after the fact and plot red circles around the tracts of interest.

Selecting the tracts with a high proportion of bike commuters, the `points` command can then be used similarly to the `plot` command. However, `points` draws on top of the current plot rather than creating a new graphic window; it also does not accept parameters which describe the plot as a whole such as axes limits (`ylim`) and titles (`main`).

```
> index <- which(walkPerc > 0.3)
> points(hhIncome[index, "30k"] / hhIncome[index, "total"],
+        hhIncome[index, "200k"] / hhIncome[index, "total"],
+        col="red")
```

The resulting Fig. 4.5 now clearly calls out the location of these high-bike commuter tracts.

## 4.5 Line Plots

The next, and final, dataset from the American Community Survey that we investigate calculates the time of day in which people typically leave their house for work. Like the other datasets, we convert this to a matrix; we also immediately convert the raw counts into proportions using the “total” column.

```
> timeOfCommute <- read.csv("data/ch03/timeOfCommute.csv",
+                           as.is=TRUE, check.names=FALSE)
> timeOfCommute <- as.matrix(timeOfCommute)
> timeOfCommute[, -1] <- timeOfCommute[, -1] / timeOfCommute[, 1]
```

The columns represent time ranges, with the labels giving the end of the time range. So, the column labeled “5am” gives the proportion of the population who leave for work between midnight and “5am”.

```
> timeOfCommute[1:5,]
      total      5am      5:30am      6am      6:30am      7am
[1,]  1094 0.05850091 0.053016453 0.05210238 0.12888483 0.11060329
[2,]  1122 0.03297683 0.070409982 0.04188948 0.08288770 0.14171123
[3,]   886 0.02708804 0.049661400 0.11060948 0.06546275 0.05530474
[4,]  1001 0.07492507 0.006993007 0.02397602 0.04795205 0.06593407
[5,]  1054 0.03036053 0.004743833 0.01612903 0.09392789 0.03415560
      7:30am      8am      8:30am      9am      10am      11am
[1,] 0.17093236 0.1755027 0.05667276 0.06032907 0.04753199 0.03747715
[2,] 0.11408200 0.2638146 0.06862745 0.02852050 0.03119430 0.03208556
[3,] 0.14785553 0.1591422 0.14672686 0.03611738 0.05079007 0.00000000
[4,] 0.13586414 0.3336663 0.07892108 0.06193806 0.06193806 0.00000000
[5,] 0.09962049 0.1764706 0.16129032 0.06925996 0.01707780 0.10056926
      12pm      4pm      12am
[1,] 0.00000000 0.02010969 0.0283363803
[2,] 0.018716578 0.07219251 0.0008912656
[3,] 0.012415350 0.02821670 0.1106094808
[4,] 0.000000000 0.06993007 0.0379620380
[5,] 0.008538899 0.06831120 0.1195445920
```

In this case, we may want to visualize just a single row of data by creating a scatter plot of time of day against the proportion of the population departing for work at that hour. To start with this requires (manually in this case) determining numeric values which write the column name hours as a numeric variable in hours past midnight.

```
> numericTimes <- c(5,5.5,6,6.5,7,7.5,8,8.5,9,10,11,12,16,24)
> plot(numericTimes, timeOfCommute[1,-1])
```

The points in this plot would actually be better represented as a single line passing through each point. To do this in R, the `lines` function is used; the syntax is similar to `text` and `points`.

```
> lines(numericTimes, timeOfCommute[1,-1])
```

The output of this one tract’s data is shown in Fig. 4.6.

One problem with our initial plot is that we are using raw proportions even though the time buckets are not evenly distributed. For instance, the last column represents 8 h of the day (4 p.m.-midnight), and it is unfair to compare this count to a bucket with only half an hour of time. To alleviate this we construct a modified version of the time data where each bucket is divided by the number of hours it contains.

```
timeOfCommuteDens <- timeOfCommute[, -1]
timeOfCommuteDens[,1] <- timeOfCommuteDens[,1] / 5
timeOfCommuteDens[,2:9] <- timeOfCommuteDens[,2:9] / 0.5
timeOfCommuteDens[,13] <- timeOfCommuteDens[,13] / 4
timeOfCommuteDens[,14] <- timeOfCommuteDens[,14] / 8
```

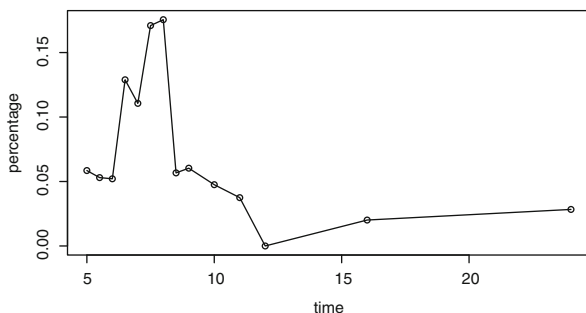


Figure 4.6: Raw values from a particular census tract, showing the proportion of the population who leave at a given hour (shown as hours from midnight).

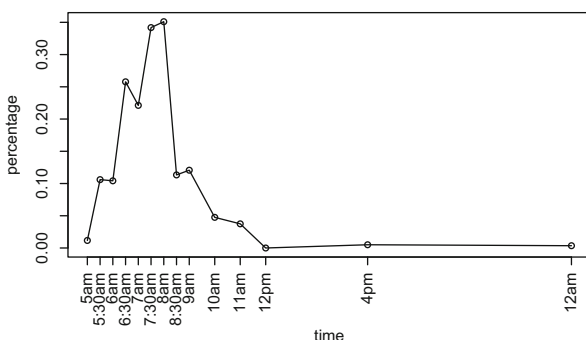


Figure 4.7: Line plot showing the density of people who commute to work at a given hour, from one particular tract.

Using these **densities** rather than raw values will allow for a more accurate description of commute times.

The second issue with the original plot is the difficulty in reading the x-axis labels. This is supposed to represent time, but in converting to a numeric variable for plotting this is not reflected in the axes. To fix these, we must manually construct the axis labels. We have already seen how to turn off the default axes using the `axes` parameter and manually adding back the box around the plot.

```
> plot(numericTimes, timeOfCommuteDens[1,], axes=FALSE)
> lines(numericTimes, timeOfCommuteDens[1,])
> box()
```

To add custom axes to a plot, the `axis` function is used. With a single number, the default axis will be added to the corresponding side.<sup>2</sup> As the y-axis was fine in the

<sup>2</sup>1: bottom, 2: left, 3: top, 4: right.

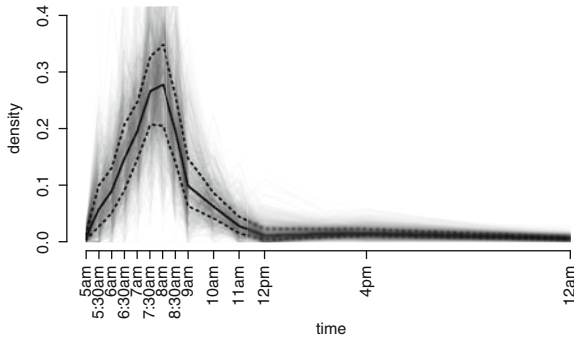


Figure 4.8: Density of time commuters leave for work. The *solid line* shows the hourly median and *dashed lines* indicate the first and third quantiles.

original plot, we use `axis(2)` to add it back in. For the x-axis, we can specify the location of the labels by the `at` parameter and the labels by the `label` parameter.

```
> axis(2)
> axis(1, at=numericTimes, label=colnames(timeOfCommuteDens), las=2)
```

The `las=2` is added to rotate the labels so that they do not run into one another. The improved plot is shown in Fig. 4.7; notice how much easier the plot is to read, and how the densities in the large buckets on the extreme ends have decreased.

Now that we have a nice plot of a single census tract, how might we similarly represent all of the tracts? A separate plot for each would be difficult to use as there are over 800 tracts. A better solution is to use a for loop to plot all of the lines in the same graphics window. If we did this verbatim, the large number of lines would mostly just form a hard-to-read black mass of points. The trick is to use an opaque color for the lines by specifying the `alpha` value in the `grey` function. This fills the lines with a see-through color. Setting the `alpha` to a number close to zero creatively turns the simple line function into a way to visualize all of the tracts on the same plot.

```
> plot(numericTimes, timeOfCommuteDens[1,], type="n", axes=FALSE,
+      xlab="time", ylab="density", ylim=c(0,0.4))
> for(j in 1:nrow(timeOfCommuteDens)) {
+   lines(numericTimes, timeOfCommuteDens[j,], col=grey(0,alpha=0.01))
+ }
```

The output of this is shown in Fig. 4.8, along with the following additional three lines which plot the median, first, and third quantiles over the plot.

```
> medianTimes <- apply(timeOfCommuteDens,2,quantile,probs=0.5)
> lines(numericTimes, medianTimes)
> q1Times <- apply(timeOfCommuteDens,2,quantile,probs=0.25)
> lines(numericTimes, q1Times, col=rgb(0,0,0), lty="dashed")
```



```
> q3Times <- apply(timeOfCommuteDens, 2, quantile, probs=0.75)
> lines(numericTimes, q3Times, col=rgb(0,0,0), lty="dashed")
```

From this we see that 8 a.m. is the most popular time to leave, followed very closely by 7:30 a.m. The peak drops off sharply at 9 a.m., at which point most commuters have already left. There is a steadily increasing trickle of early commuters from 5 a.m. to the peak hours.

## 4.6 Scatter Plot Matrix

We have shown how to construct a scatter plot to visualize the relationship between two continuous variables. By utilizing shapes, colors, sizes, lines, text, and points, we have also been able to incorporate additional variables into a single plot. These are all great techniques, particularly when producing final plots to succinctly present information in a paper, book, or presentation. When dealing with a large set of data, particularly when seeing it for the first time, a less clever solution is often best for understanding the general relationship between the variables. A common tool uses a matrix of scatter plots, showing the relationships between all **pairs** of continuous variables, as an initial visualization tool. The R function **pairs** creates such a plot from just a data frame or matrix; in order to illustrate several important concepts, we will show how to instead construct such a graphic manually.

We start by creating a four column dataset of combining selected variables from the means of transportation, time of commute, and income datasets.

```
> tractData <- data.frame(walkPerc, carPerc,
+                         inc30k=timeOfCommuteDens[, "7am"],
+                         inc200k=hhIncome[, "200k"]/hhIncome[, "total"])
```

It turns out that one row of this dataset has a missing value in it. In order to remove this, we combine the **is.na** function from Sect. 2.9 and apply function from Sect. 3.9, picking out only rows with no missing values.

```
> tractData <- tractData[apply(is.na(tractData), 1, sum) == 0, ]
```

We have previously had this missing value floating around in our dataset, but up until now we have not needed to remove it because the **scatter plot functions handle missing** values by simply ignoring them.

In order to construct the scatter plot of matrices, we need to construct *nested* for loops, where one loop is contained inside another. We also need to use a new control flow function **if**, which evaluates its argument, and only if true execute the remainder of the code. For example, the following code cycles through all combinations of *i* and *j* in the set 1 to 5, but prints the result only when *i* is strictly less than *j*.

```
> for (i in 1:5) {
+   for (j in 1:5) {
+     if (i < j) print(paste(i, ":", j))
+   }
+ }
```

```
+   }
+ }
[1] "1 : 2"
[1] "1 : 3"
[1] "1 : 4"
[1] "1 : 5"
[1] "2 : 3"
[1] "2 : 4"
[1] "2 : 5"
[1] "3 : 4"
[1] "3 : 5"
[1] "4 : 5"
```

These nested loops and control statements can become complex, but should be understandable by decomposing them into their respective parts.

The matrix of scatter plot requires setting three graphical parameters. We have already seen how to tell R to put multiple plots on a single image using `par` and to change the plot margins using `mar`. For this plot we want to have small margins between the plots, but a larger margin around the entire graphic; this is achieved by manipulating the `oma` parameter.

```
> par(mfrow=c(4,4))
> par(mar=c(1,1,1,1))
> par(oma=c(2,2,2,2))
```

With these in place, we now cycle through all combinations of the variables `i` and `j` to create the matrix of scatter plots. The `if` statement is used to: (1) add a set of vertical axis labels down the first column, (2) add a set of horizontal axes labels across the bottom row, and (3) add a title using the `title` function to the top row of the matrix. Opacity has been used to help show overlapping points.

```
> par(mfrow=c(4,4))
> par(mar=c(1,1,1,1))
> par(oma=c(2,2,2,2))
> for (i in 1:ncol(tractData)) {
+   for (j in 1:ncol(tractData)) {
+     plot(tractData[,j], tractData[,i], pch=19, col=grey(0,0.2),
+         axes=FALSE)
+     box()
+     if (i == 1) title(main=colnames(tractData)[j])
+     if (i == ncol(tractData)) axis(1)
+     if (j == 1) axis(2)
+   }
+ }
```

This code constructs the graphic shown in Fig. 4.9. It provides a simple way of visualizing the entire data frame at once. Many useful tweaks can be applied to our basic plot. The diagonal of the plot is not particularly insightful, for example, and could be replaced with a histogram of the associated variable. Also, the top and bottom of the grid are simple mirrors of one another and one copy could be replaced

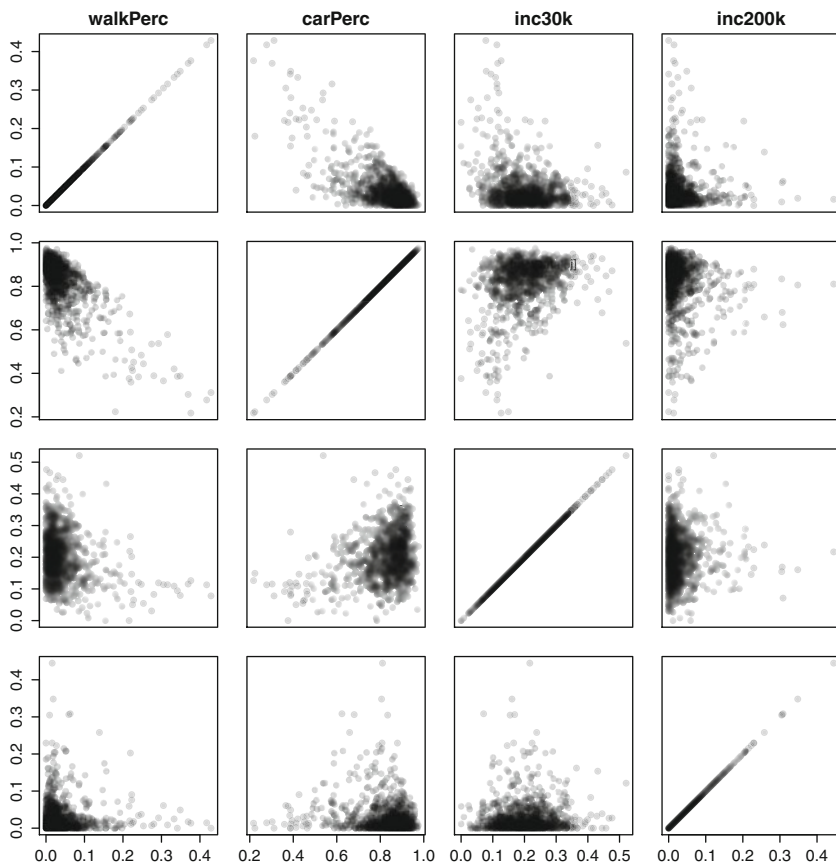


Figure 4.9: Manually constructed pairs plot of four derived variables from the American Community Survey for tracts in the state of Oregon.


with alternative information or graphics. For examples of these alternatives, see the help pages for `pairs`.

## 4.7 Correlation Matrix

We have focused on simple tables and graphical methods for exploratory data analysis, avoiding model-based approaches such as t-tests and qq-plots. One numerical technique which deserves at least a passing mention are correlation matrices. The correlation between two variables is a number between  $-1$  and  $1$  describing the strength of the linear relationship between them.<sup>3</sup> A value close to  $1$  indicates that one variable is approximately a positive multiple of the other. For example, the number of households and the population in a census tract have a correlation of

<sup>3</sup>Technically this is known as the Pearson correlation, but when “correlation” is used without a qualifier this is generally what is meant.

0.92. Negative correlations close to  $-1$  indicate a similar relationship, but where one variable tends to increase as the other decreases. Correlations near zero indicate the lack of linear relationship between the variables in question.

A correlation matrix calculates all pairs of correlations in a dataset, analogous to a scatter plot matrix. Calculating a correlation matrix in R is as easy as calling the function `cor`  numeric matrix.

```
> corMat <- cor(tractData)
> round(corMat, 2)
```

	walkPerc	carPerc	inc30k	inc200k
walkPerc	1.00	-0.71	-0.27	-0.04
carPerc	-0.71	1.00	0.30	-0.13
inc30k	-0.27	0.30	1.00	0.02
inc200k	-0.04	-0.13	0.02	1.00

Unsurprisingly, the percentage of people who walk to work and the percentage who drive to work are negatively correlated. When one proportion is high, we would naturally expect the other to (generally) be low. The two income band counts are relatively uncorrelated, something we saw at the county level in Fig. 4.3. It does seem that tracts with a higher than typical proportion of car commuters have more households in the \$30,000 income bracket but a decreased number of high earners.

## Chapter 5

### EDA III: Advanced Graphics



**Abstract** In this chapter, we show several methods for increasing the usability and aesthetic quality of graphics in R. Random number generators and color spaces are also introduced as tools for creating quality graphics.

#### 5.1 Introduction

The graphics we constructed in the previous two chapters have done a nice job of allowing us to quickly explore and understand categorical and numeric variables. We used graphical parameters such as color to represent additional information in scatter plots, while tweaking margins, layouts, and color opacity to fit multiple analyses into a single graphics window. The resulting visualizations already generally look clean and professional. The flexibility of the R programming language provides tools for further increasing the aesthetic value of these plots; this chapter explores several of these methods. As a by-product, the resulting graphics are also often better from the standpoint of basic data exploration.

#### 5.2 Output Formats

In order to use R graphics for presentations and publications, we need a way of saving them as external files. There are often several ways to do this such as right-clicking the plot or selecting from a drop-down menu depending on the method of accessing R. These approaches are fine for quickly saving results during an exploratory analysis. It is also possible to save the graphical output to a file by calling a sequence of R functions. We recommend this approach as it is platform and console independent. If you have a long script that produces dozens of graphics, updating all of the graphics is just a matter of copying and pasting the entire script into R. Finally, parameters such as the width and height of the plot are fixed in the code

and will not change from run to run. When using the menu-based options, these parameters are usually determined by the current window size (and is very difficult to replicate).

To save R graphics as a *portable document format* (pdf) file, the function `pdf` is called *prior* to calling any of the graphics commands. The filename of the output as well as width and height of the output in inches should typically all be specified.<sup>1</sup>

```
> pdf(file="filename.pdf", width=4, height=4)
```



After calling this command, functions such as `plot` and `text`, which previously created graphics windows inside of R, will now only plot to the pdf file. If multiple commands are called that would have overwritten a previous plot, these are placed in a new page of the pdf file (another benefit of the command over point and click methods). Once all of the graphics command have been executed, we need to **tell R to close its connection to** the pdf file by turning the device off. This is done with the command `dev.off()`; it should return a short reference to the device to let us know that it has been successfully closed.<sup>2</sup>

```
> dev.off()
null device
      1
```

After closing the pdf file, graphics commands will work as normal with results displayed directly in R windows. The resulting pdf file can be opened using any external pdf viewer and embedded into e-mails, presentations, websites, and other media. Do not try to open the file prior to closing the graphics device, as it will appear corrupted and will not be viewable.

Other output formats can be saved using a similar method. The functions `bmp`, `jpeg`, `png`, and `tiff` each save files in formats corresponding to their names. These differ from the `pdf` function in that the default height and width are in pixels (individual points) rather than inches, and they do not support multiple pages. If you plot more than one graphics window with these, only the last graphic appears in the final output. The following code illustrates how to save a scatter plot to a png file:

```
> png(filename="myScatterPlot.png", height=800, width=800)
> plot(1:10, 1:10)
> dev.off()
null device
      1
```

While the `pdf` command saves vector images, the four functions—`bmp`, `jpeg`, `png`, and `tiff`—save raster images (hence the reason behind the slightly different default values). The distinction between these two file formats, and relative pros and

<sup>1</sup> Otherwise a plot named "Rplots.pdf" will be created in the current working directory with a width and height of 7 inches.

<sup>2</sup> The device should also close if you exit the R session entirely, but it is a better practice to manually close it so as to not risk corrupting the file.

cons of each, is explored at length throughout Chap. 7 in the context of geospatial datasets and again in Chap. 8 when exploring image analysis.

When constructing complex plots, such as the large networks in Chap. 6, plotting can become a painfully slow operation. Writing the output to a file rather than directly to an R windows is often significantly faster; in these cases, it can be advantageous to use these graphics commands even during the initial stages of data exploration.

### 5.3 Color

Good use of color is an essential part of creating most statistical graphics. We have seen how to use colors by name and through the command `grey`. Here we show how to easily construct a *color palette*, a small vector of the colors used in a plot, for various tasks. To illustrate these palettes, we load a data set of election results from the 2012 presidential election in France.

```
> election <- read.csv("../raw_data/france_election_2012.csv",
+                       as.is=TRUE)
> election[1:5,]
```

	department	HOLLANDE	SARKOZY	LE.PEN	MELENCHON	BAYROU	JOLY
1	Paris	34.83	32.19	6.20	11.09	9.34	4.18
2	Seine-et-Marne	27.65	27.27	19.65	11.01	8.55	1.96
3	Yvelines	27.32	34.24	12.44	9.11	11.24	2.50
4	Essonne	30.39	25.46	15.20	12.26	9.33	2.35
5	Hauts-de-Seine	30.16	34.97	8.51	10.35	10.69	2.74

	DUPONT.AIGNAN	POUTOU	ARTHAUD	CHEMINADE	HOLLANDE_2	SARKOZY_2
1	1.00	0.67	0.27	0.23	55.60	44.40
2	2.18	1.02	0.46	0.26	49.25	50.75
3	1.73	0.80	0.35	0.28	45.70	54.30
4	3.41	0.94	0.41	0.24	53.43	46.57
5	1.34	0.69	0.30	0.26	49.48	50.52

Each row corresponds to the results from a particular department, an administrative division. Columns 2–11 give the percentage of votes won in the first round of voting, with columns 12 and 13 giving the percentage of votes in the second round of voting (when only two candidates were on the ballot).

One variable of interest from the first round of voting is the total percentage of votes from each department that went to candidates other than the two finalists, François Hollande and Nicolas Sarkozy. Using the `apply` function this is calculated as follows:

```
> otherVotes <- apply(election[,4:11],1,sum)
```

From the binning method developed in Sect. 3.5, these votes can be grouped into ten buckets.

```
> cuts <- quantile(otherVotes,probs=seq(0,1,length.out=11))
> bins <- cut(otherVotes, cuts, include.lowest=TRUE, labels=FALSE)
```

The vector `bins` has numbers 1 to 10, matching the `otherVotes` variable to a discrete categorization of buckets.

We want to map each of these ten buckets to a set of colors. Specifically, we need a *sequential palette* to show a gradual change between two extremes. We use the R package **colorspace** to provide this set of colors. For details on installing R packages, see Chap. 11. The function `heat_hcl`, provided by the package, returns a set of colors ranging from a dark red to a light yellow. The number of colors used to span this set is given as an input parameter, here we use ten to match the number of buckets.

```
> library(colorspace)
> heat_hcl(n=10)
[1] "#D33F6A" "#DA565E" "#E06B50" "#E57E41" "#E8913" #EAA428"
[7] "#E9B62D" "#E8C842" "#E5D961" "#E2E6BD"
```

The colors are represented as hexadecimal codes, which is also the output given by `grey`, and can be used directly by the R plotting functions.<sup>3</sup> As it is more common to have red by the highest value, and yellow the lowest, we reverse this set before saving it as our color palette. These colors are then used to construct a vector `cols` from the bins.

```
> heatPalette = heat_hcl(10)[10:1]
> cols = heatPalette[bins]
```

Now the vector `cols` can be used to encode the percentage of the vote taken by the remaining parties in a scatter plot showing the percentages given to the top two candidates.

```
> plot(election$HOLLANDE, election$SARKOZY, col=cols, pch=19,
+       cex=1.5, xlab="HOLLANDE %", ylab="SARKOZY %")
```

The result is shown in Fig. 5.1a. As expected, the reddest points occur in the lower left of the plot (where Hollande and Sarkozy did relatively poorly) and the yellowest points occur in the upper right half of the plot.<sup>4</sup>

Given the two stage election method in France, one interesting metric is to look at how voters for other candidates split their votes in the second round.<sup>5</sup> For example, we can compute the percentage of the remaining votes assigned to François Hollande as follows:

<sup>3</sup>Several websites, such as <http://www.color-hex.com/>, provide a quick way of decoding these into a thumbnail of the color.

<sup>4</sup>The coloring is determined entirely by the other two axes, so this works great as an example of what a sequential palette of heat colors looks like. It does not, however, provide much useful new information to the scatter plot.

<sup>5</sup>We are implicitly assuming that voters for the front-runners are not changing their votes and that the exact same group of people vote in each round.



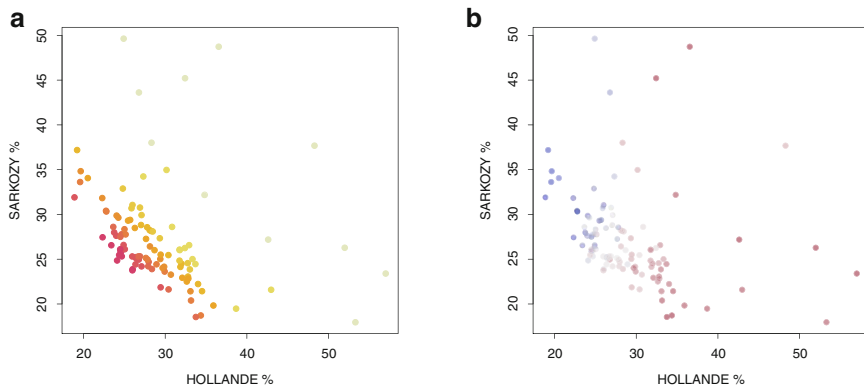


Figure 5.1: **(a)** Scatter plot of first round 2012 French Presidential Elections data, illustrating a sequential color palette. **(b)** Scatter plot of first round 2012 French Presidential Elections data, illustrating a divergent color palette.

```
> percRemain <- (election$HOLLANDE_2 - election$HOLLANDE) /
+ (100 - election$HOLLANDE - election$SARKOZY)
> quantile(percRemain)
      0%      25%      50%      75%     100%
0.3613821 0.4817990 0.5251723 0.5768840 0.8498641
```

By symmetry the remainder of the votes were allotted to Sarkozy. Consider assigning these percentages to colors in the same way we did with the percentages of voters casting votes for candidates not proceeding to the second round. The basic idea would still be conveyed, but the color scheme from dark red to light yellow is not particularly accurate because this data has a natural mid-point at 50 %. Interesting values are not those at one end of the spectrum, but rather values at either extreme. A better choice would be a palette ranging from a dark color, through white, and back to an alternative color. White should correspond to an even 50–50 split of the remaining votes.

To construct the bins to be used with such a color palette, we have to calculate the data cuts separately for values less than and greater than 50. Otherwise there is no guarantee that both of these sets will have the same number of buckets; we need this property so that the “white” value in the middle corresponds to an even split. To calculate these cuts, we use the `quantile` function on only a subset of the data to construct upper and lower break points.

```
> index <- percRemain < 0.5
> cutsLower <-
+   quantile(percRemain[index], probs=seq(0,1,length.out=11))
> index <- percRemain >= 0.5
> cutsUpper <-
+   quantile(percRemain[index], probs=seq(0,1,length.out=11))
```

These breakpoints are combined together into a single vector and used, as before, to bin the output data into chunks.

```
> cuts = c(cutsLower,cutsUpper)
> cuts
      0%      10%      20%      30%      40%      50%      60%
0.3613821 0.4161632 0.4340408 0.4527129 0.4627394 0.4721817 0.4772551
      70%      80%      90%     100%      0%      10%      20%
0.4818796 0.4859240 0.4926577 0.4978733 0.5001059 0.5125271 0.5231833
      30%      40%      50%      60%      70%      80%      90%
0.5351469 0.5482774 0.5606758 0.5760597 0.5874583 0.6174553 0.6376925
      100%
0.8498641
> bins = cut(percRemain, cuts, include.lowest=TRUE, labels=FALSE)
```

The **colorspace** package provides a function `diverge_hcl` that gives a *divergent palette* of colors. It uses a white value as its mid-point, diverging toward two separate colors on either end point. Here we add an additional parameter `alpha` to make the colors opaque, and assign colors to each data point using the palette and data bins.

```
> divPalette <- diverge_hcl(20,alpha=0.5)
> cols <- divPalette[bins]
```

The scatter plot of first round results for the two leading candidates using this new color scale is shown in Fig. 5.1b. Here the colors do in fact add new information to the original plot. We see in general that Hollande did a better job of picking up votes in the second round, largely the reason he ultimately won the election. For the most part each candidate did better swaying voters in area where they were already the stronger of the two, though there were two, shown in the upper center, where Sarkozy won the first round but Hollande gained significant ground in the second round.

One interesting way of categorizing the various departments is to see which of the losing first-round candidates had the highest showing. We calculate this by again using the `apply` function together with a new function `which.max`, which returns the index of the position with the highest value.

```
> whichOtherParty <- apply(election[,4:11],1,which.max)
> table(whichMaxThirdParty)
whichMaxThirdParty
 1  2  3
94  7  6
```

We see that Front National candidate Marine Le Pen, identified with index 1 by looking at the candidate names in the original data, dominated over all the other candidates by taking 94 of the 107 departments. Parti de Gauche candidate Jean-Luc Mélenchon and the centrist François Bayrou managed to split the remaining 13.

In order to color points by these categories, we need a categorical color palette where each value is highly distinguishable. This final category is a *categorical*

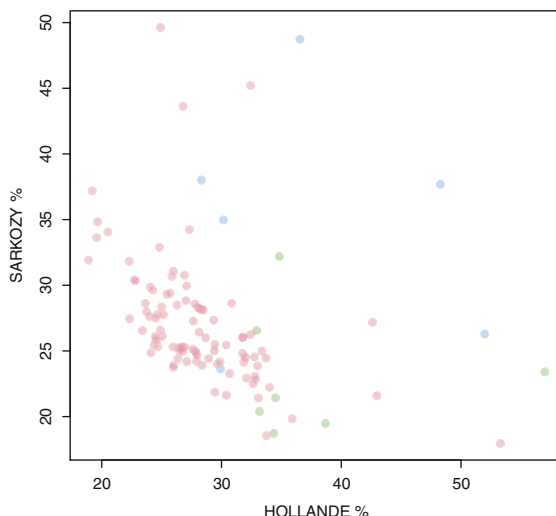


Figure 5.2: Scatter plot of first round 2012 French Presidential Elections data, illustrating a categorical color palette based on the runner-up with the best results by department.

*palette* and can be calculated by the `rainbow_hcl` function. It is possible to construct this type of palette by hand, but this function will typically do a better job of finding aesthetically pleasing colors which are maximally differentiated.<sup>6</sup> When dealing with categorical variables which have many levels, hand selecting colors also becomes increasingly difficult.

```
> categoricalPalette <- rainbow_hcl(3, alpha=0.5)
> cols <- categoricalPalette[whichMaxThirdParty]
```

The plot from this is shown in Fig. 5.2. Pink, being the most dominant, is obviously Le Pen, with Mélenchon is the greenish one and Bayrou the blue. We see that the centrist Bayrou performed well when Hollande and Sarkozy captured a high amount of the initial vote. Mélenchon on the other hand did well where Hollande was also strong; this is reasonable given that Parti de Gauche was a recent off-shoot of Hollande's Parti Socialiste.

All of these palette creation routines have additional parameters for controlling the output (e.g., the divergent palette can be made to instead range between red and green). See the excellent vignette which comes along with the package for additional details.

<sup>6</sup>In this particular case hand constructing colors may be a better alternative as we could pick those associated with each party. We use the `colorspace` function to illustrate the more general approach.

## 5.4 Legends

The **colorspace** package has given us great color palettes for various types of binned and categorical data. A missing element has been a legend for indicating the meaning behind the values. These can be added to R plots via the `legend` command, which takes the x and y coordinates where it should be plotted and a vector of labels corresponding to the elements in the legend. Additional graphical parameters affect the shape, color, and size of the points plotted in the legend. Here is an example of the legend that would be used in the plot of the runner-ups in the first round of the 2012 French Presidential election:

```
> legend("topright", legend=colnames(election)[4:6],
+       col=categoricalPalette, pch=19, cex=1,
+       bg=grey(0.9))
```

Figure 5.3 shows the much improved plot with the new legend. We used the `bg` parameter to place a light grey background on the image.

A more involved legend can be used to display the levels in a sequential palette. Recomputing the colors for the maximum number of other votes,

```
> otherVotes <- apply(election[,4:11], 1, sum)
> cuts <- quantile(otherVotes, probs=seq(0,1,length.out=11))
> bins <- cut(otherVotes, cuts, include.lowest=TRUE, labels=FALSE)
> cols <- heatPalette[bins]
```

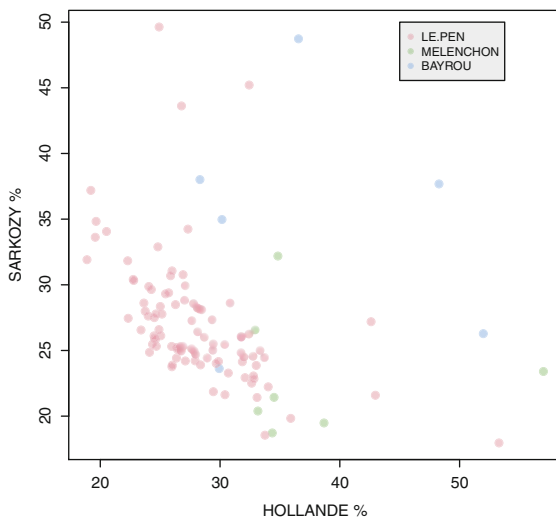


Figure 5.3: Scatter plot of first round 2012 French Presidential Elections data, showing the use of a categorical legend.

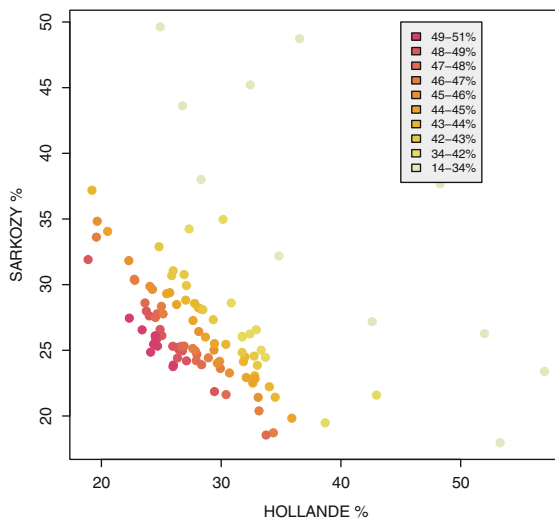


Figure 5.4: Scatter plot of first round 2012 French Presidential Elections data, showing the use of a legend with a continuous response.

We take a rounded version of the break points and paste them together for the legend labels.

```
> cuts <- round(cuts[11:1])
> legendLabels <- paste(cuts[-1], "-", cuts[-length(cuts)], "%", sep="")
```

The legend of the plot is now just as before with the new labels and colors.

```
> legend(45, 50, legend=legendLabels, fill=heatPalette, bg=grey(0.9))
```

The output in Fig. 5.4 displays the plot with a legend, which now allows us to give scale to the colors.

## 5.5 Randomness

When looking at the department by which of the remaining parties did the best, we noticed several interesting patterns but were not able to easily identify the identity of the interesting data points. Using the `text` command could alleviate this by displaying names next to data points. Here we set the points color to “white”, so that only the text values are displayed.

```
> cols <- categoricalPalette[whichOtherParty]
> plot(election$HOLLANDE, election$SARKOZY, col="white",
+       pch=19, cex=2, xlim=c(10,60), ylim=c(10,60))
> text(election$HOLLANDE, election$SARKOZY, election$department,
+       col=cols)
```

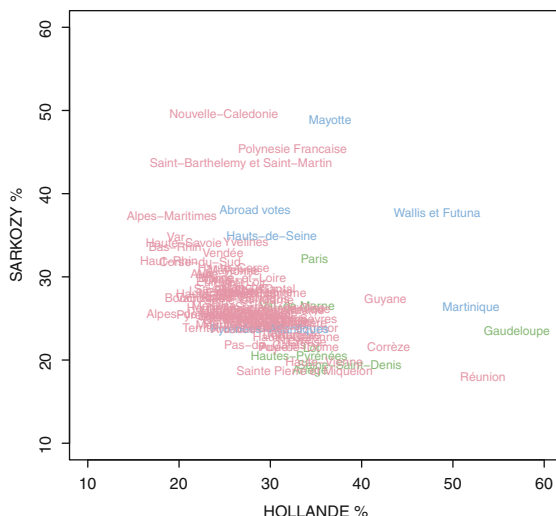


Figure 5.5: Scatter plot of first round 2012 French Presidential Elections data, giving department names.

Unfortunately, the resulting Fig. 5.5 has many overlapping text boxes making the majority of them unreadable. For our own personal use, a simple solution is to make the text size significantly smaller and they just zoom in on each region. This solution is neither elegant nor practical when using the image in print media or as a presentation tool.

As a tool for statistical analysis, the R language has an extensive number of functions for generating and working with randomly generated values.<sup>7</sup> We will not need most of these, but will make use of the function `sample`, which takes a vector and returns a randomly chosen subset of the vector of a given length.

```
> sample(1:10, 3)
[1] 8 6 9
```

Running this command many times will yield new results. By default, the sample only picks a value from the set once; the ordering of the output is also randomly generated.

We can use this command to pick out only a random subset of the data to plot. Due to the importance in our visualization of the 13 departments where Le Pen was not the best runner-up, we take a subset consisting of those 13 data points and 15 randomly selected others.

<sup>7</sup>Technically R only has the ability to construct *pseudorandom* numbers, but conceptually for most purposes the distinction is unimportant.

```
> index <- which(whichOtherParty %in% c(2,3))
> index <- c(index, sample(which(whichOtherParty == 1), 15))
```

We now create the plot again, but only for the subset of points in index.

```
> plot(election$HOLLANDE, election$SARKOZY, col="white",
+       pch=19, cex=2, xlim=c(10,60), ylim=c(10,60),
+       xlab="HOLLANDE %", ylab="SARKOZY %")
> text(election$HOLLANDE[index], election$SARKOZY[index],
+       election$department[index], col=cols[index], cex=0.6)
```

As we see in Fig. 5.6, this is only a modest improvement. We have lost many of the data points which were previously not a problem, but still have over-plotting in the dense middle of the plot because the randomly selected set is likely to still pick several data points from that region.

A clever solution to this problem is to assign every data point to a number giving its approximate location in the plot. Consider the following scheme:

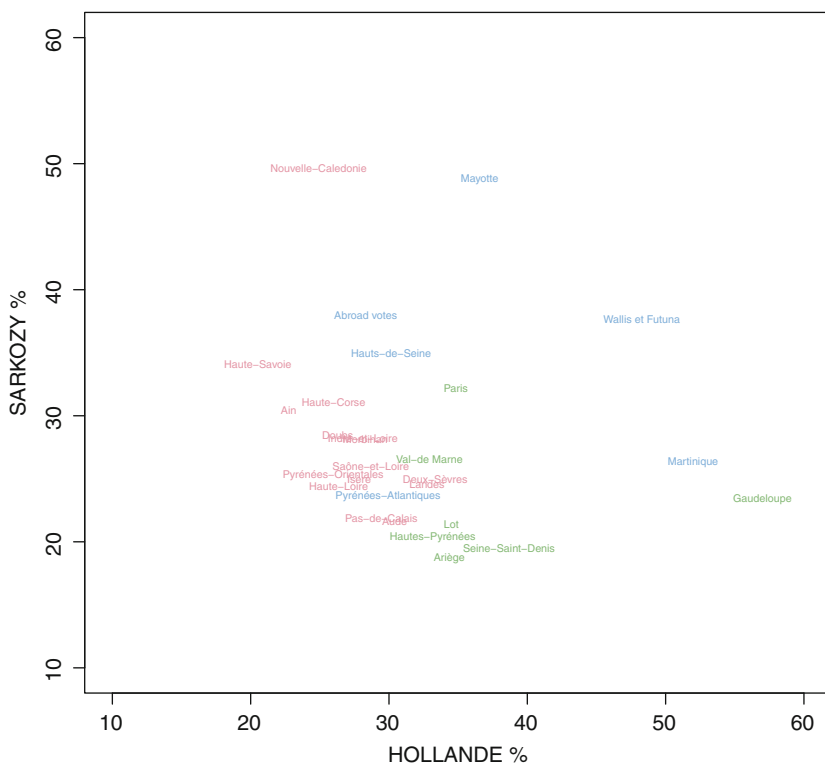


Figure 5.6: Scatter plot of first round 2012 French Presidential Elections data, giving department names. Random sampling used to limit overplotting.

```

> location <- round(election$SARKOZY/2) +
+   round(election$HOLLANDE/10)*100
> location
 [1] 316 314 317 313 317 410 313 313 312 215 215 214 312 213 312 214
[17] 313 312 315 312 314 314 315 314 314 215 314 311 313 214 312 311
[33] 312 213 213 213 217 216 314 213 313 312 313 315 315 313 216 312
[49] 312 313 314 312 314 313 312 311 312 312 313 312 309 313 311 312
[65] 311 310 312 313 411 311 410 215 312 313 312 313 315 214 217 312
[81] 314 312 311 311 212 313 314 313 213 213 219 214 217 214 216 316
[97] 612 513 414 509 309 322 424 519 323 225 319

```

The first two digits give the approximate number of votes assigned to Sarkozy, to a precision of 2, and the hundreds digit gives the votes assigned to Hollande to a precision of 10. Two points now have the same location value only if they gave similar votes to each candidate. If we allow only one text label to appear for any given location value, the plot should not have any intersecting labels; we used a longer box for the horizontal axis because text takes more horizontal than vertical space.<sup>8</sup>

Also, this time we plot a very opaque version of all the data as a base layer; this means that no data points are completely missing, even though some will be lacking a textual label.

```

> colsAlpha <- rainbow_hcl(3, alpha=0.2)[whichMaxOtherParty]
> plot(election$HOLLANDE, election$SARKOZY, pch=19, cex=2,
+      xlim=c(10,60), ylim=c(10,60),
+      xlab="HOLLANDE %", ylab="SARKOZY %")

```

We now use a *for loop* to cycle through the data, plotting a text label only if we have not seen a point with that given location value. Because this would unfairly prioritize the rows of data near the top of our data frame, we use the `sample` command to randomly permute the indices.<sup>9</sup>

```

> index <- c()
> for (i in sample(1:nrow(election))) {
+   if (!(location[i] %in% index)) {
+     text(election$HOLLANDE[i], election$SARKOZY[i]+1,
+          election$department[i], col=cols[i], cex=0.7)
+     index = c(index, location[i])
+   }
+ }

```

Figure 5.7 shows a very professional looking plot with many of the points labeled and others only have lightly shaded dots. In particular, notice that all of the points outside the densely populated center are completely labeled as they share a location value with no other points.

<sup>8</sup>If this scheme seems confusing, try running the code yourself and tweaking the 2 and 10 values. It is significantly easier to describe this in code and examples than via a textual description.

<sup>9</sup>If `sample` is used without a sample size, it returns a random permutation of the entire input. This is the same as would occur when the sample size is manually set to the length of the input.



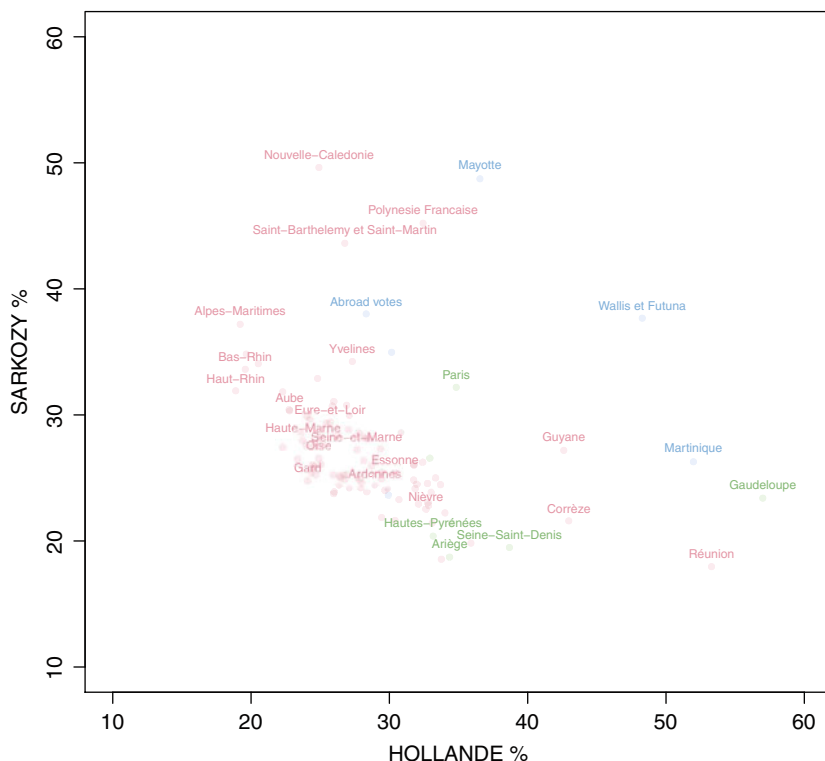


Figure 5.7: Scatter plot of first round 2012 French Presidential Elections data, giving department names. A grid-base version of sampling is used to limit overplotting of the text boxes.

When using the `sample` command, we expect the results to change with each call. If this were not the case, the whole purpose of the function would be lost. Sometimes, however, we want to make sure that a block of R code using a random sample will pick the same sample each time it is run. For example, in this text we wanted to make sure that each plot looks the same when we update the code so that our description of the output does not also need to be updated. In order to fix the result of subsequent calls, the function `set.seed` is used and given an arbitrary integer at which to set itself. The following code should produce the same subsets every time it is run.

```
> set.seed(42)
> sample(1:10,3)
[1] 10 9 3
> sample(1:10,3)
[1] 9 6 5
```

Notice that we called `sample` twice and received two different results. We mention this primarily as it is used throughout the supplementary material so that readers working along with the code can exactly duplicate our results when they depend on a randomly chosen subset.

## 5.6 Additional Parameters

Typing `par()` in an R session prints out 72 parameters that effect the output of an R plot. Dozens of other parameters are set directly within the plot function and the calls to the graphics devices such as `pdf` and `jpeg`. We will not take the time to explain the massive amount of customization that is ultimately possible from within R.<sup>10</sup> We have covered some of the most important and difficult ones already in this and the proceeding two chapters. Many data type specific plotting parameters are mentioned throughout the remainder of this text. We touch briefly on two additional commands which accomplish tasks which are often asked by new users of R graphics.

The plain white background of R plot windows are nice for print media, but can be a bit boring for digital uses. Adding a grey background requires only one line of code, though it is not a particularly intuitive piece of code:

```
> rect(par("usr")[1], par("usr")[3], par("usr")[2], par("usr")[4],
+      col = gray(0.9))
```

To understand what this is doing look at the examples in the help pages for `rect` and `usr`. It can otherwise be copied and pasted as is to add colorful background to existing plots. With white freed from being a background color, it can now be used to provide grid lines for the final plot using the `grid` command.

```
> grid(lty="solid", col="white")
```

The result of adding a background and grid to our French election data is shown in Fig. 5.8. Note that we had to make a separate call to the `points` after adding the background and grid as these had over-writing the initial plot.

In busy plots, it is often useful to put the legend outside of the main graphics window. In order to make this work, the `margin` command must first be used to create additional margin space outside the plot window.

```
> par(mar=c(5.1, 4.1, 4.1, 10))
> plot(election$HOLLANDE, election$SARKOZY, col=cols, pch=19,
+      cex=1.5, xlab="HOLLANDE %", ylab="SARKOZY %")
```

The `legend` command is then used with the additional parameter `xpd=TRUE` (which tells R that it is okay to let it plot outside the main window). The  $x$  and  $y$  coordinates are specified assuming they continue outside of the plot window. Here we use  $x = 60$  because it falls just to the right of the plot.

<sup>10</sup>For a good description to many of these additional parameter, see Paul Murrell's text *R graphics* [1].

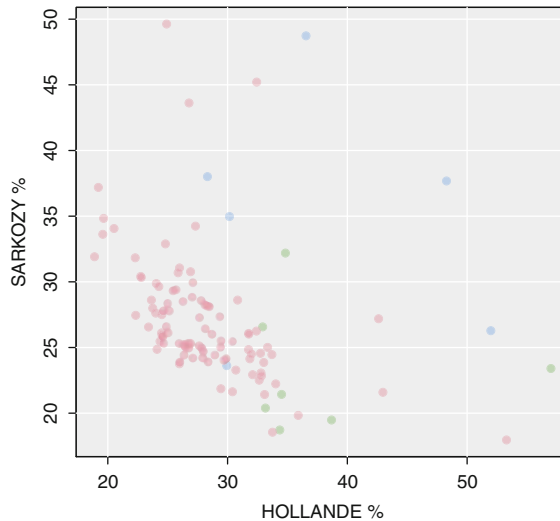


Figure 5.8: Scatter plot of a grey background with white grid lines. Closely mimics the default style of **ggplot2**.

```
> legend(60,50,legend=colnames(election)[4:6],
+       col=rainbow_hcl(3),pch=19,cex=1,
+       xpd=TRUE,
+       bty = "n")
```

This legend outside the plot is shown in Fig. 5.9. When plots have a large number of levels, or span the entire region, this trick can be particularly helpful.

## 5.7 Alternative Methods

The style of visualization we have been using, and will continue to use, are called *base R* graphics. We find them to be the easiest to work with for new users, while allowing for a wide range of customization for more advanced work. Several alternatives with completely different plotting commands and constructs do exist and are also quite popular. The packages **lattice** and **grid** are shipped with the standard installation of R and each have their own constructs for plotting. The third-party package **ggplot2** has an even further modified set of functions for plotting, built around Leland Wilkinson's text *Grammar of Graphics* [2, 3]. We mention these as they are frequently mentioned in various help forums and mailing lists.

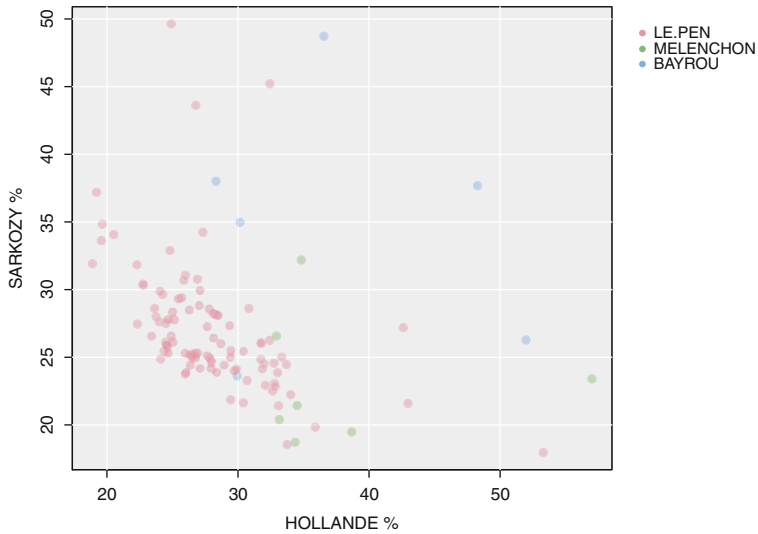


Figure 5.9: Example showing a legend outside the main plotting region.

## References

- [1] Paul Murrell. *R graphics*. CRC Press, 2011.
- [2] Hadley Wickham. *ggplot2: elegant graphics for data analysis*. Springer Science & Business Media, 2009.
- [3] Leland Wilkinson, D Wills, D Rope, A Norton, and R Dubbs. *The grammar of graphics*. Springer Science & Business Media, 2006.

## Chapter 12

### 100 Basic Programming Exercises

**Abstract** In this chapter, short programming exercises based on the material from Chaps. 2 to 5 are presented.

The following questions should be solvable using material directly presented in the respective chapters, with two notable exceptions. We did not introduce the concept of user-generated functions or how to capture answers to questions from the terminal. These were not directly applicable to the flow of the introductory chapters, but make for great building blocks for interesting programming questions. The following code snippet can be used as a template for how to do both:

```
> ask <- function() {  
>   z <- readline("enter your name: ")  
>   print(paste("Hello ", z, "!", sep=""))  
> }  
> ask()
```

When running the last line, R will ask for your name. After hitting “enter”, the message will be displayed. Notice that the result from `readline` is always a character vector. If the result should be a number it must be explicitly converted.



```
> ask <- function() {  
>   z <- readline("enter a number: ")  
>   z <- as.numeric(z)  
>   return(z + 1)  
> }  
> ask()
```

The other expectation is the use of the `if` statement throughout questions in Chaps. 2 and 3, even though it was not formally introduced until Sect. 4.6. The `if` statement evaluates the statement after it only if the argument is true. For example, the following prints only one of the two statement:

```
> if (1 > 2) print("one is larger than two")
> if (1 < 2) print("one is less than two")
[1] "one is less than two"
```

Using this construct for questions the earlier chapters allowed us to significantly widen their scope and extent without overly complicating the material.

## Chapter 2

1. Ask for a positive number and return a vector of all the numbers between 1 and the input.
2. Ask for a number and return a vector of all the even numbers between 1 and the input.
3. Ask for a positive integer  $n$ . Return the sum:  $1 + 1/2 + 1/3 + 1/4 + \dots + 1/n$ .
4. Ask for the total of a bill and return the amount of a 15 % tip. The `round` function is useful for cleaning up the result to an even penny.
5. Ask for a user's birth year and print the age they will turn this year. You can write the current year directly without trying to determine it externally (it is possible to determine the current year, but was not covered in Chap. 2).
6. Write a function which asks for a number and determines if it is a whole number (or not). Print a message displaying the result using `print`.
7. The factorial of an integer is the product of all the positive integers less than or equal to it. For example, the factorial of 4 is equal to  $4 * 3 * 2 * 1 = 24$ . There is a function `factorial` in base R for calculating these. Ask for a number and  the factorial, without using the R function. Hint: The function `prod`  be helpful.
8. Ask the user for a number between 1 and 10 and return the corresponding simple ordinal number. For example, 1 should be 1st, 2 should be 2nd, and 4 should be 4th. Hint: You should not write 10 separate if statements. Notice that the numbers 4–10 all have the same ending of “th”.
9. Repeat the previous question, but allow the user to input any whole, positive number. Hint: Keep the input as a character vector and make use of the `nchar` and `substr` functions.

10. There is a character vector available in base R called `state.abb` giving the two digit postal abbreviations for the 50 US States. Write a function which asks for an abbreviation and returns `TRUE` if it is an abbreviation and `FALSE` otherwise.
11. Repeat the previous question, but allow for cases where the user inputs a different capitalization. For example, “CA”, “Ca”, and “ca” should all return as `TRUE`.
12. R provides another vector of state names as a vector called `state.name`. The elements line up with the abbreviations; for example, element 33 of the abbreviations is “NC” and element 33 of the names is “North Carolina”. Write a function which asks for an abbreviation and returns the corresponding state name. If there is none, return the string “error”.
13. Finally, write a function which asks for either a state name or state abbreviation. When given an abbreviation, it returns the state name, and when given a name it returns the state abbreviation. If there is no match to either it returns the string “error”.
14. The object `state.x77` is a matrix that gives several summary statistics for each of the 50 US States from 1977. Calculate the number of high school graduates in each state, and sort from highest to lowest.
15. Calculate the number of high school graduate per square mile in each state.
16. Ask the user to provide a state abbreviation, and return the number of high school graduates in that state in 1977.
17. Now, print a vector of the state names from the highest illiteracy rate to the lowest illiteracy rate. Hint: The state names are given as `rownames`.
18. Construct a data frame with ten rows and three columns: `Illiteracy`, `Life_expectancy`, `Murder`, and `HS_grad`. Each column gives the names of the worst 10 state names. Hint: some measures are good when they are high and others are good when they are low. You will need to take account of these.
19. Using vector notation, print the state names which are in both the top 10 for illiteracy and top 10 for murder rates in 1977.
20. There are several other small datasets contained within the base installation of R. One of these is the Titanic dataset, accessed by the object `Titanic`. The format is a bit strange at first, but can be converted to a data frame with the following code:

```
ti <- as.data.frame(Titanic)
```

It has a row for each combination of Class, Sex, Age, and Survival flag, along with a frequency count (see `?Titanic` for more information). Write a function which asks the user to input a Class category (either “1st”, “2nd”, “3rd”, or “Crew”) and prints the **total** survival and death counts for that category.


21. Take the titanic dataset and again ask the user to select a class. Write the subset of the data from this class and save it as a comma separated value file named “titanicOutput.csv” in the current working directory. Print to the user the full path of the created file.
22. Ask the user for the working directory where the previous command was run. Set the working directory to this location, read the titanic dataset into R and return it.
23. Repeat the previous question, but instead print the passenger Class for which the file “titanicOutput.csv” was saved.
24. R provides another dataset called `WorldPhones` giving the number of telephones in seven world regions, in thousands, for the years 1951, 1956, 1957, 1958, 1959, 1960, and 1961. Calculate the percentage change in number of phones for each region between 1951 and 1961. Use vector notation, do not do each region by hand! Hint: Percentage change is  $(\text{new value} - \text{old value})/\text{old value}$ .
25. Ask the user for a year between 1951 and 1961. Return the number of phones in Europe for the year closest, but not after, the input year.

## Chapter 3

26. The dataset `iris` is a very well-known statistical data from the 1930s. It gives several measurements of iris sepal and petal lengths (in centimeters) for three species. Construct a table of sepal length rounded to the nearest centimeter versus Species.
27. Construct the same table, but rounded to the nearest half centimeter.
28. Plot a histogram of the sepal length for the entire iris dataset.
29. Replicate the previous histogram, but manually specify the break points for the histogram and add a custom title and axis labels.
30. Plot three histograms showing the sepal length separately for each species. Make **sure** the histograms use the same break points for each plot (Hint: use the same as manually set in the previous question). Add helpful titles for the plots, and make sure to set R to display three plots at once.
31. Calculate the deciles of the petal length for the entire iris dataset.





32. Construct a table showing the number of samples from each species with petal length in the top 30 % of the dataset. How well does this help categorize the dataset by species?
33. Now bin the iris dataset into deciles based on the petal length. Produce a table by species. How well does this categorize the dataset by species?
34. We can get a very rough estimate of the petal area by multiplying the petal length and width. Calculate this area, bin the dataset into deciles on area, and compute table of the petal length deciles against the area deciles. How similar do these measurements seem?
35. Without using a for loop, construct a vector with the median petal length for each species. Add appropriate names to the output.
36. Repeat the previous question using a for loop.
37. Finally, repeat again using `tapply`.
38. As in a previous question, write a function which asks the user for a state abbreviation and returns the state name. However, this time, put the question in a for loop so the user can decode three straight state abbreviations.
39. The command `break` immediately exits a for loop; it is often used inside of an `if` statement. Redo the previous question, but break out of the loop when a non-matching abbreviation is given. You can increase the number of iterations to something large (say, 100), as a user can always get out of the function by giving a non-abbreviation.
40. Now, reverse the process so that the function returns when an abbreviation is found but asks again if it is not.
41. Using a for loop, print the sum  $1 + 1/2 + 1/3 + 1/4 + \dots + 1/n$  for all  $n$  equal to 1 through 100.
42. Now calculate the sum for all 100 values of  $n$  using a single function call. 
43. Ask the user for their year of birth and print out the age they turned for every year between then and now.
44. The dataset `InsectSprays` shows the count of insects after applying one of six different insect repellents. Construct a two-row three-column grid of histograms, on the same scale, showing the number of insects from each spray. Do this using a for loop rather than coding each plot by hand.
45. Repeat the same two by three plot, but now remove the margins, axes, and labels. Replace these by adding the spray identifier (a single letter) to the plot with the `text` command.
46. Calculate the median insect count for each spray.

47. Using the `WorldPhones` dataset, calculate the total number of phones used in each year using a for loop.
48. Calculate the total number of phones used in each year using a single apply function.
49. Calculate the percentage of phones that were in Europe over the years in question.
50. Convert the entire `WorldPhones` matrix to percentages; in other words, each row should sum to 100.

## Chapter 4

51. Produce a scatter plot of sepal length versus petal length in the iris dataset.
52. Add color to the scatter plot of sepal length versus petal length to distinguish the three iris species. Use solid dots to highlight the colors.
53. Using the previous plot as a starting point, change the size of the points based on the sepal width. Hint: A good way to get nice sizes is to divide the sepal width by the median width.
54. Change the plot to use the text of the species type instead of dots. You can remove the sizing based on sepal width so that all of the text is of the same size; however, it may help to make all the text points smaller than the default to reduce overplotting.
55. Add a vertical and horizontal median to the scatter plot of sepal length versus petal length.
56. Add *by group* vertical and horizontal medians to the scatter plot of sepal length versus petal length. There should be three vertical and three horizontal lines. Color the lines and points based on species, and make the lines dashed rather than solid.
57. Reconstruct the scatter plot of sepal length versus petal length with species colors. Add text to show the medians of the three groups. Test out different sizes for the points and text to make nice looking plot.
58. Construct a plot of petal length versus sepal width. The plot has two large clusters of points (in the upper left and lower right) and one outlier in the bottom left. Construct a coloring for these three groups. Produce a side by side plot showing petal length versus sepal width next to sepal length versus petal length. Use the new color scheme for both.
59. Create a scatter plot matrix (pairs plot) from the four numerical variables in the iris dataset. You should reduce the margins to fit the entire plot on the screen but do not need to directly replicate all of the tweaks from Sect. 4.6.

60. Change the custom scatter plot matrix to have a histogram on the diagonal axis.
61. Take the `InsectRepelant` dataset and produce a single line plot of the data. It will be easiest going forward to do this by (1) plotting the data with a white color, and (2) making a single call to the `lines` function.
62. The first row of the `InsectRepelant` dataset has a count of 10. Add to the plot a stack of solid points to represent this count between the coordinates (1,1) and (1,10).
63. Now, replicate this for every row of data on a plot. Do not include the line as it will no longer be needed. You can do this in several ways, though the most straightforward is as a loop over the rows of the dataset. Hint: You may need to create a special case for the two rows with a count of zero.
64. Duplicate the previous plot, but use the `spray` type as a text object (it is a single letter between A and F) in place of the dots).
65. Add vertical bars to separate the groups. Do not do this manually, but use vectors to plot all the lines at once.
66. Using dots again instead of letters replicate the plot such that one dot represents three counts. Do not plot any remainder, so a count of 5 should have only one dot. Hint: The `floor` function will be helpful; it removes the fractional part of a number.
67. Now, redo the plot such that the fractional part of the remainder is represented by a smaller dot. So a remainder of 0.33 gets a dot with `cex=0.33` and a remainder of 0.66 gets a dot with `cex=0.66`.
68. The object `AirPassengers` is a dataset which gives the number of international airline passengers by month for 12 years of data. It is stored by default in an atypical format but can be converted to a matrix easily:

```
ap <- matrix(as.numeric(AirPassengers), ncol=12)
rownames(ap) <- month.abb
colnames(ap) <- 1949:1960
```

Calculate the total flyers for each year, and then calculate the total number of flyers over each month. Are there any noticeable patterns?

69. Construct a line plot of the number of fliers for 1949. Use custom axes to label the months.
70. Construct a graphic with line plots for every year. Add a text label to the data point for July to label the years. Hint: Use `range(ap)` to determine the limits of the y-axis to capture all of the data.

71. Produce a scatter plot of the 1949 data against the 1950 data. Use `text` to label the points. Remember to offset the labels from the points.
72. Produce a new dataset which shows the month percentage of flights for each year. In other words, each column should sum to 100. Save this as `scaledAp`.
73. Create a new line chart showing these standardized values for each year over the months.
74. Create an empty plot with x and y ranges from 1 to 12. Create a dot at each coordinate  $(i, j)$  to represent the value in the 12 by 12 matrix `ap`. Use relative sizes to show the value. Construct useful custom axes (use the option `las=2` to make the axes look nicer).
75. Recreate the plot using the `scaledAp` data. Color points blue if they are less than 100/12 and red if they are greater than 100/12 (we use this cut-off as it represents a typical month).

## Chapter 5

76. The dataset `ChickWeight` contains several time series showing the weight of young chickens feed one of four different diets. Produce a series of line plots using categorical colors. Use the **colorspace** package for the colors.
77. Remove the five chicks with incomplete data. Normalize the weights of each chick such that each weighs 0 on the first day and 1 on the last day.
78. Construct a matrix with one row for each chick showing the growth rate, (new weight—old weight/old weight) for each day. This will have 11 columns and a row for each chick.
79. Plot the chick ids against the growth rates.
80. Change labels from the previous plot into text commands giving the day in time for a random sample set of 3 days selected for each chick.
81. Repeat the previous plot with solid dots, where a sequential palette is used to indicate time.
82. Create and place a legend decoding the colors from the previous plot. You will have to increase the size of the plot and perhaps make the legend smaller than the default to fit everything. What do you notice about typical growth rates from this plot now?
83. Take `AirPassengers` dataset and plot the 1949 counts against the 1950 counts, using a categorical palette to distinguish the four seasons of the data.
84. Redo the previous plot using a sequential heat palette to show the months and include a legend to explain the colors.

85. Plot two randomly chosen numeric variables from the iris dataset.
86. A previous question asked to plot the sepal length versus petal length for the iris data, where each point is labeled with text giving the plant species. Redo this plot using subsampling to reduce the overplotting. Use a color palette to select the colors for the plot.
87. Now, redo the plot of sepal length versus petal length using a sequential color palette to display sepal width, using five buckets for the bins.
88. Add a helpful legend for the colors in the previous plot to describe the coloring of the points.
89. Reconstruct the `scaledAp` data, but this time subtract the 100/12 from every entry. Create a histogram of the results. The average value should be zero, but the median of the scaled results will be negative.
90. Create a divergent palette with 21 bins from the `scaledAp`. Recall that this requires two sets of bins. Use this to plot the entire dataset as a single time series; add a line through the data points to improve readability.
91. Edit the previous line plot to have a legend off to the right of the entire plot, and add custom x-axis labels showing the years.
92. Write a for loop that cycles over the numbers 1–100. For each iteration of the loop, if the number is divisible by 3, print “fizz” to the console, and if divisible by 5, print “buzz”. Otherwise print the number itself. (This is a well-known intro interview question for new programmers called FizzBuzz).
93. Repeat FizzBuzz using a vector in place of a loop. The return value will be a character vector.
94. Create a function which asks the user for how many letters they would like their password to be and then generates a random string of letters/numbers of that length as a random password. Hint: R has the objects `letter` and `LETTERS` built in a default object.
95. Sample 1000 points from the set containing just 1 and  $-1$ . Calculate the cumulative sum of this sample. This simulates a mathematical model known as a random walk.
96. Plot the cumulative sum of the random walk over time as a line plot. Run it several times to see different random outputs. What does the plot remind you of?
97. R provides the function `Sys.sleep`, which causes the program to wait for a given number of seconds before proceeding. Combine this with a short time period (0.1 seconds is a good first guess) to animate the previous plot by waiting in-between drawing each line segment.

98. Calculate 10,000 separate random walks, saving the 500th and 1000th step from each. Take the result and create a scatter plot of the results; use a color with an alpha channel to reduce the effects of over plotting. What patterns do you see in the plot?
99. Calculate side-by-side histograms of the 500th and 1000th positions from the previous simulation. What patterns do you notice here?
100. Write a function to simulate the game “rock-scissors-paper”. You should ask the user to select one of these by name, generate a random response, and indicate who won.