

## Chapter 2

# First Foray into Text Analysis with R

**Abstract** In this chapter readers learn how to load, tokenize, and search a text. Several methods for exploring word frequencies and lexical makeup are introduced. The exercise at the end introduces the `plot` function.

### 2.1 Loading the First Text File

If you have not already done so, set the working directory to the location of the supporting materials directory.<sup>1</sup>

```
setwd("~/Documents/TextAnalysisWithR")
```

You can now load the first text file using the `scan` function<sup>2</sup>:

```
text.v <- scan("data/plainText/melville.txt", what="character", sep="\n")
```

Type this command into a new R script file and then run it by either copying and pasting into the console or using RStudio's control and return shortcut. This is as good a place as any to mention that the `scan` function can also load files from the Internet. If you have an Internet connection, you can enter a url in place of the file path and load a file directly from the web, like this:

---

<sup>1</sup> Programming code is extremely finicky. If you do not type the commands exactly as they appear here, you will likely get an error. In my experience about 95% of the errors and bugs one sees when coding are the result of careless typing. If the program is not responding the way you expect or if you are getting errors, check your typing. Everything counts: capital letters must be consistently capitalized, commas between arguments must be out side of the quotes and so on.

<sup>2</sup> Throughout this book I will use a naming convention when instantiating new R objects. In the example seen here, I have named the object `text.v`. The `.v` extension is a convention I have adopted to indicate the R data type of the object, in this case a *vector* object. This will make more sense as you learn about R's different data types. For now, just understand that you can name R objects in ways that will make sense to you as a human reader of the code.

```
text.v <- scan("http://www.gutenberg.org/cache/epub/2701/pg2701.txt",
              what="character", sep="\n")
```

Whether you load the file from your own system—as you will do for the exercises in this book—or from the Internet, if the code has executed correctly, you should see the following result:

```
Read 18874 items
>
```

Remember that the `>` symbol seen here is simply a new R prompt indicating that R has completed its operation and is ready for the next command. At the new prompt, enter:

```
> text.v
```

You will see the entire text of *Moby Dick* flash before your eyes.<sup>3</sup> Now try:

```
> text.v[1]
```

You will see the contents of the first *item* in the `text.v` variable, as follows<sup>4</sup>:

```
[1] "The Project Gutenberg EBook of Moby Dick;
or The Whale, by Herman Melville"
```

When you used the `scan` function, you included an *argument* (`sep`) that told the `scan` function to separate the file using `\n`. `\n` is a *regular expression* or *meta-character* (that is, a kind of computer shorthand) representing (or standing in for) the newline (carriage return) characters in a text file. What this means is that when the `scan` function loaded the text, it broke the text up into chunks according to where it found newlines in the text.<sup>5</sup> These chunks were then stored in what is called a *vector*, or more specifically a *character vector*.<sup>6</sup> In this single R expression, you invoked the `scan` function and put the results of that invocation into a new object named `text.v`, and the `text.v` object is an object of the type known as a character vector.<sup>7</sup> Deep breath.

It is important to understand that the data inside this new `text.v` object is *indexed*. Each line from the original text file has its own special container inside the `text.v` object, and each container is numbered. You can access lines from the

<sup>3</sup> Actually, you may only see the first 10,000 lines. That's because R has set the `max.print` option to 10,000 by default. If you wish to increase the default for a given work session, just begin your session by entering `options(max.print=1000000)`.

<sup>4</sup> From this point forward, I will not show the R prompt in the code examples.

<sup>5</sup> Be careful not to confuse newline with “sentence” break or even with paragraph. Also note that depending on your computing environment, there may be differences between how your machine interprets `\n`, `\r` and `\r\n`.

<sup>6</sup> If you have programmed in another language, you may be familiar with this kind of data structure as an *array*.

<sup>7</sup> In this book and in my own coding I have found it convenient to append suffixes to my variable names that indicate the type of data being stored in the variable. So, for example, `text.v` has a `.v` suffix to indicate that the variable is a vector: `v = vector`. Later you will see *data frame* variables with `.df` extensions and lists with a `.l`, and so on.

original file by referencing their *container* or *index* number it within a set of square brackets. Entering

```
text.v[1]
```

returns the contents of the first container, in this case, the first line of the text, that is, the first part of the text file you loaded up to the first newline character. If you enter `text.v[2]`, you'll retrieve the second chunk, that is, the chunk between the first and second newline characters.<sup>8</sup>

## 2.2 Separate Content from Metadata

The electronic text that you just loaded into `text.v` is the plain text version of *Moby Dick* that is available from *Project Gutenberg*. Along with the text of the novel, however, you also get a lot of *Project Gutenberg* boilerplate metadata. Since you do not want to analyze the boilerplate, you need to determine where the novel begins and ends. As it happens, the main text of the novel begins at line 408 and ends at 18576. One way to figure this out is to visually inspect the output you saw (above) when you typed the object name `text.v` at the R prompt and hit return. If you had lightning fast eyes, you might have noticed that `text.v[408]` contained the text string: "CHAPTER 1. Loomings." and `text.v[18576]` contained the string "orphan." Instead of removing all this boilerplate text in advance, I opted to leave it so that we might explore ways of accessing the items of a character vector.

Let's assume that you did not already know about items 408 and 18576, but that you did know that the first line of the book contained "CHAPTER 1. Loomings" and that the last line of the book contained "orphan." We could use this information along with R's `which` function to isolate the main text of the novel. To do this on your own, enter the following code, but be advised that in R, and any other programming language for that matter, accuracy counts. The great majority of the errors you will encounter as you write and test your first programs will be the results of careless typing.<sup>9</sup>

```
start.v <- which(text.v == "CHAPTER 1. Loomings.")
end.v <- which(text.v == "orphan.")
```

In reality, of course, you are not likely to know in advance the exact contents of the items that `scan` created by locating the *newline* characters in the file, and the `which` function requires that you identify an exact match. Later on I'll show a better way of handling this situation using the `grep` function. For now just pretend that you know where all the lines begin and end. You can now see the line numbers

---

<sup>8</sup> Those who have programming experience with another language may find it disorienting that R (like FORTRAN) begins indexing at 1 and not 0 like C, JAVA and many other languages.

<sup>9</sup> In these expressions, the two equal signs serve as a comparison operator. You cannot use a single equals sign because the equals sign can also be used in R as an *assignment* operator. I'll clarify this point later, so for now just know that you need to use two equals signs to compare values.

(the vector indices) returned by the `which` function by entering the following at the prompt<sup>10</sup>:

```
start.v
end.v
```

You should now see the following returned from R:<sup>11</sup>

```
start.v
## [1] 408
end.v
## [1] 18576
```

In a moment we will use this information to separate the main text of the novel from the metadata, but first a bit more about character vectors...

When you loaded *Moby Dick* into the `text.v` object, you asked R to divide the text (or delimit it) using the carriage return or *newline* character, which was represented using `\n`. To see how many newlines there are in the `text.v` variable, and, thus, how many items, use the `length` function:

```
length(text.v)
## [1] 18874
```

You'll see that there are 18,874 lines of text in the file. But not all of these lines, or *strings*, of text are part of the actual novel that you wish to analyze.<sup>12</sup> *Project Gutenberg* files come with some baggage, and so you will want to remove the non-*Moby Dick* material and keep the story: everything from "Call me Ishmael..." to "...orphan." You need to reduce the contents of the `text.v` variable to just the lines of the novel proper. Rather than throwing out the metadata contained in the *Project Gutenberg* boilerplate, save it to a new variable called `metadata.v`, and then keep the text of the novel itself in a new variable called `novel.lines.v`. Do this using the following four lines of code:

```
start.metadata.v <- text.v[1:start.v -1]
end.metadata.v <- text.v[(end.v+1):length(text.v)]
metadata.v <- c(start.metadata.v, end.metadata.v)
novel.lines.v <- text.v[start.v:end.v]
```

Entering these commands will not produce any output to the screen. The first line of code takes lines 1 through 407 from the `text.v` variable and puts them into a new variable called `metadata.v`. If you are wondering where the 407 came from, remember that the `start.v` variable you created earlier contains the value 408 and refers to a place in the text vector that contains the words "CHAPTER 1. Loomings." Since you want to keep that line of the text (that is, it is not metadata but part of the novel) you must subtract 1 from the value inside the `start.v` variable to get the 407.

<sup>10</sup> In R you can enter more than one command on a single line by separating the commands with a semi-colon. Entering `start.v;end.v` would achieve the same result as what you see here.

<sup>11</sup> In this book the output, or results, of entering an expression will be prefaced with two hash (##) symbol.

<sup>12</sup> Sentences, lines of text, etc. are formally referred to as *strings* of text.

The second line does something similar; it grabs all of the lines of text that appear after the end of the novel by first adding one to the value contained in the `end.v` variable then spanning all the way to the last value in the vector, which can be calculated using `length`.

The third line then combines the two using the `c` or *combine* function. As it happens, I could have saved a bit of typing by using a shortcut. The same result could have been achieved in one expression, like this:

```
metadata.v <- c(text.v[1:(start.v-1)], text.v[(end.v+1):length(text.v)])
```

Sometimes this kind of shortcutting can save a lot of extra typing and extra code. At the same time, you will want to be careful about too much of this function *embedding* as it can also make your code harder to read later on.

The fourth line of the main code block is used to isolate the part of the electronic file that is between the metadata sections. To capture the metadata that comes after the last line in the novel, I use `(end.v + 1)` because I do not want to keep the last line of the novel which is accessed at the point in the vector currently stored in the `end.v` variable. If this does not make sense, try entering the following code to see just what is what:

```
text.v[start.v]
text.v[start.v-1]
text.v[end.v]
text.v[end.v+1]
```

You can now compare the size of the original file to the size of the new `novel.lines.v` variable that excludes the boilerplate metadata:

```
length(text.v)
## [1] 18874
length(novel.lines.v)
## [1] 18169
```

If you want, you can even use a little subtraction to calculate how much smaller the new object is: `length(text.v) - length(novel.lines.v)`.

The main text of *Moby Dick* is now in an object titled `novel.lines.v`, but the text is still not quite in the format you need for further processing. Right now the contents of `novel.lines.v` are spread over 18,169 line items derived from the original decision to delimit the file using the newline character. Sometimes, it is important to maintain line breaks: for example, some literary texts are encoded with purposeful line breaks representing the lines in the original print publication of the book or sometimes the breaks are for lines of poetry. For our purposes here, maintaining the line breaks is not important, so you will get rid of them using the `paste` function to *join* and *collapse* all the lines into one long string:

```
novel.v <- paste(novel.lines.v, collapse=" ")
```

The `paste` function with the `collapse` argument provides a way of *gluing* together a bunch of separate *pieces* using a *glue character* that you define as the value for the `collapse` argument. In this case, you are going to glue together the lines (the *pieces*) using a blank space character (the *glue*). After entering this expression,

you will have the entire contents of the novel stored as a single string of words, or rather a string of characters. You can check the size of the novel object by typing

```
length(novel.v)
## [1] 1
```

At first you might be surprised to see that the length is now 1. The variable called `novel.v` is a vector just like `novel.lines.v`, but instead of having an indexed slot for each line, it has just one slot in which the entire text is held. If you are not clear about this, try entering:

```
novel.v[1]
```

A lot of text is going to flash before your eyes, but if you were able to scroll up in your console window to where you entered the command, you would see something like this:

```
[1] "CHAPTER 1. Loomings. Call me Ishmael. Some years ago..."
```

R has dumped the entire contents of the novel into the console. Go ahead, read the book!

## 2.3 Reprocessing the Content

Now that you have the novel loaded as a single string of characters, you are ready to have some fun. First use the `tolower` function to convert the entire text to lowercase characters.

```
novel.lower.v <- tolower(novel.v)
```

You now have a big blob of *Moby Dick* in a single, lowercase string, and this string includes all the letters, numbers, and marks of punctuation in the novel. For the time being, we will focus on the words, so you need to extract them out of the full text string and put them into a nice organized list. R provides an aptly named function for splitting text strings: `strsplit`.

```
moby.words.l <- strsplit(novel.lower.v, "\\W")
```

The `strsplit` function, as used here, takes two arguments and returns what R calls a *list*.<sup>13</sup> The first argument is the object (`novel.lower.v`) that you want to split, and the second argument `\\W` is a *regular expression*. A *regular expression* is a special type of character string that is used to represent a pattern. In this case, the regular expression will match any non-word character.<sup>14</sup> Using this simple *regex*, `strsplit` can detect *word boundaries*.

So far we have been working with vectors. Now you have a list. Both vectors and lists are data types, and R, like other programming languages, has other data types

<sup>13</sup> Because this new object is a list, I have appended “.l” to the variable name.

<sup>14</sup> WIKIPEDIA provides a fairly good overview of regular expression and a web search for “regular expressions” will turn up all manner of useful information.

as well. At times you may forget what kind of data type one of your variables is, and since the operations you can perform on different R objects depends on what kind of data they contain, you may find yourself needing the `class` function. R's `class` function returns information about the data type of an object you provide as an argument to the function. Here is an example that you can try:

```
class(novel.lower.v)
## [1] "character"
class(moby.words.l)
## [1] "list"
```

To get even more detail about a given object, you can use R's `str` or *structure* function. This function provides a compact display of the internal structure of an R object. If you ask R to give you the structure of the `moby.words.l` list, you'll see the following:

```
str(moby.words.l)
## List of 1
## $ : chr [1:253989] "chapter" "1" "" "loomings" ...
```

R is telling you that this object (`moby.words.l`) is a list with one item and that the one item is a character (`chr`) vector with 253989 items. R then shows you the first few items, which happen to be the first few words of the novel.<sup>15</sup> If you look closely, you'll see that the third item in the `chr` vector is an empty string. We'll deal with that in a moment. ...

Right now, though, you may be asking, why a list? The short answer is that the `strsplit` function that you used to split the novel into words returns its results as a list. The long answer is that sometimes the object being given to the `strsplit` function is more complicated than a simple character string and so `strsplit` is designed to deal with more complicated situations. A list is a special type of object in R. You can think of a list as being like a file cabinet. Each drawer is an item in the list and each drawer can contain different kinds of objects. In my file cabinet, for example, I have three drawers full of file folders and one full of old diskettes, CDs and miscellaneous hard drives. You will learn more about lists as we go on.

It is worth mentioning here that anytime you want some good technical reading about the nature of R's functions, just enter the function name preceded by a question mark, e.g.: `?strsplit`. This is how you access R's built in "help" files.<sup>16</sup> Be forewarned that your mileage with R-help may vary. Some functions are very well documented and others are like reading tea leaves.<sup>17</sup> One might be tempted to blame

<sup>15</sup> I have adopted a convention of appending a data type abbreviation to the end of my object names. In the long run this saves me from having to check my variables using `class` and `str`.

<sup>16</sup> Note that in RStudio there is a window pane with a "help" tab where the resulting information will be returned. This window pane also has a search box where you can enter search terms instead of entering them in the console pane.

<sup>17</sup> `?functionName` is a shortcut for R's more verbose `help(functionName)`. If you want to see an example of how a function is used, you can try `example(functionName)`. `args(functionName)` will display a list of arguments that a given function takes. Finally, if you want to search R's documentation for a single keyword or phrase, try using `??("your keyword")` which is a shorthand version of `help.search("your keyword")`. I wish I could say that the documentation in R is always brilliant; I can't. It is inevitable that as you learn more about R you will find places where the

poor documentation on the fact that R is open source, but I think it's more accurate to say that the documentation assumes a degree of familiarity with programming and with statistics. `R-help` is not geared toward the novice, but, fortunately, R has now become a popular platform, and if the built-in help is not always kind to newbies, the resources that are available online have become increasingly easy to use and newbie friendly.<sup>18</sup> For the novice, the most useful part of the built-in documentation is often found in the code examples that almost always follow the more technical definitions and explanations. Be sure to read all the way down in the help files, especially if you are confused. When all else fails, or even as a first step, consider searching for answers and examples on sites such as <http://www.stackexchange.com>.

Because you used `strsplit`, you have a list, and since you do not need a list for this particular problem, we'll simplify it to a vector using the `unlist` function:

```
moby.word.v <- unlist(moby.words.l)
```

When discussing the `str` function above, I mentioned that the third item in the vector was an empty character string. Calling `str(moby.words.l)` revealed the following:

```
## List of 1
## $ : chr [1:253989] "chapter" "1" "" "loomings" ...
```

As it happens, that empty string between *l* and *loomings* is where the period character used to be. The `\\W` regular expression that you used to split the string ignored all the punctuation in the file, but then left these little blanks, as if to say, “if I’d kept the punctuation, it’d be right here.”<sup>19</sup> Since you are ignoring punctuation, at least for the time being, these blanks are a nuisance. You will want now to identify where they are in the vector and then remove them. Or more precisely, you’ll identify where they are not!

First you must figure out which items in the vector are not blanks, and for that you can use the `which` function that was introduced previously.

```
not.blanks.v <- which(moby.word.v!="")
```

---

documentation is frustratingly incomplete. In these cases, the Internet is your friend, and there is a very lively community of R users who post questions and answers on a regular basis. As with any web search, the construction of your query is something of an art form, perhaps a bit more so when it comes to R since using the letter *r* as a keyword can be frustrating.

<sup>18</sup> This was not always the case, but a recent study of the R-help user base shows that things have improved considerably. Trey Causey’s analysis “Has R-help gotten meaner over time? And what does Mancur Olson have to say about it?” is available online at <http://badhessian.org/2013/04/has-r-help-gotten-meaner-over-time-and-what-does-mancur-olson-have-to-say-about-it/>.

<sup>19</sup> There are much better, but more complicated, regular expressions that can be created for doing word tokenization. One downside to `\\W` is that it treats apostrophes as word boundaries. So the word *can’t* becomes the words *can* and *t* and *John’s* becomes *John* and *s*. These can be especially problematic if, for example, the eventual analysis is interested in negation or possession. You will learn to write a more sophisticated *regex* in Chapter 13.



Notice how the `which` function has been used in this example. `which` performs a logical test to identify those items in the `moby.word.v` that are not equal (represented by the “`!=`” operator in the expression) to blank (represented by the empty quote marks “” in the expression). If you now enter “`not.blanks.v`” into R, you will get a list of all of the *positions* in `moby.words.v` where there is not a blank. Try it:

```
not.blanks.v
```

If you tried this, you just got a screen full of numbers. Each of these numbers corresponds to a *position* in the `moby.words.v` vector where there is *not* a blank. If you scroll up to the top of this mess of numbers, you will find that the series begins like this:

```
not.blanks.v
[1] 1 2 4
```

Notice specifically that position 3 is missing. That is because the item in the third position was an empty string! If you want to see just the first few items in the `not.blanks.v` vector, try showing just the first ten items, like this:

```
not.blanks.v[1:10]
[1] 1 2 4 6 7 8 10 11 12 14
```

With the non-blanks identified, you can overwrite `moby.words.v` like this<sup>20</sup>:

```
moby.word.v <- moby.word.v[not.blanks.v]
```

Here only those items in the original `moby.words.v` that are not blanks are retained.<sup>21</sup> Just for fun, now enter:

```
moby.word.v
```

After showing you the first 99,999 words of the novel, R will give up and return a message saying something like `[[ reached getOption("max.print") - omitted 204889 entries ]]`. Even though R will not show you the entire vector, it is still worth seeing how the word data has been stored in this vector object, so you may want to try the following:

```
moby.word.v[1:10]
## [1] "chapter" "1" "loomings" "call"
## [5] "me" "ishmael" "some" "years"
## [9] "ago" "never"
```

<sup>20</sup> Overwriting an object is generally not a great idea, especially when you are writing code that you are unsure about, which is to say code that will inevitably need debugging. If you overwrite your variables, it makes it harder to debug later. Here I’m making an exception because I am certain that I’m not going to need the vector with the blanks in it.

<sup>21</sup> A shorthand version of this whole business could be written as `moby.word.v <- moby.word.v[which(moby.word.v != "")]`.

The numbers in the square brackets are the *index* numbers showing you the position in the vector where each of the words will be found. R put a bracketed number at the beginning of each row. For instance the word “chapter” is stored in the first ([1]) position in the vector and the word “ishmael” is in the sixth ([6]) position. An instance of the word “ago” is found in the ninth position and so on. If, for some reason, you wanted to know what the 99,986th word in *Moby Dick* is you could simply enter

```
moby.word.v[99986]
## [1] "by"
```

This is an important point (not that the 99,986th word is *by*). You can access any item in a vector by referencing its index. And, if you want to see more than one item, you can enter a range of index values using a *colon* such as this:

```
moby.word.v[4:6]
## [1] "call"      "me"         "ishmael"
```

Alternatively, if you know the exact positions, you can enter them directly using the *c* combination function to create a vector of positions or index values. First enter this to see how the *c* works

```
mypositions.v <- c(4,5,6)
```

Now simply combine this with the vector:

```
moby.word.v[mypositions.v]
## [1] "call"      "me"         "ishmael"
```

You can do the same thing without putting the vector of values into a new variable. Simply use the *c* function right inside the square brackets:

```
moby.word.v[c(4,5,6)]
## [1] "call"      "me"         "ishmael"
```

Admittedly, this is only useful if you already know the index positions you are interested in. But, of course, R provides a way to find the indexes by also giving access to the contents of the vector. Say, for example, you want to find all the occurrences of *whale*. For this you can use the *which* function and ask R to find *which* items in the vector satisfy the condition of being the word *whale*.

```
which(moby.word.v=="whale")
```

Go ahead and enter the line of code above. R will return the index positions where the word *whale* was found. Now remember from above that if you know the index numbers, you can find the items stored in those index positions. Before entering the next line of code, see if you can predict what will happen.

```
moby.word.v[which(moby.word.v=="whale")]
```

## 2.4 Beginning the Analysis

Putting all of the words from *Moby Dick* into a vector of words (or, more precisely, a *character* vector) provides a handy way of organizing all the words in the novel in chronological order; it also provides a foundation for some deeper quantitative analysis. You already saw how to find a word based on its position in the overall vector (the word *by* was the 99,986th word). You then saw how you could use *which* to figure out which positions in the vector contain a specific word (the word *whale*). You might also wish to know how many occurrences of the word *whale* appear in the novel. Using what you just learned, you can easily calculate the number of *whale* tokens using *length* and *which* together<sup>22</sup>:

```
length(moby.word.v[which(moby.word.v=="whale")])
## [1] 1150
```

Perhaps you would now also like to know the total number of tokens (words) in *Moby Dick*? Simple enough, just ask R for the *length* of the entire vector:

```
length(moby.word.v)
## [1] 214889
```

You can easily calculate the percentage of *whale* occurrences in the novel by dividing the number of *whale hits* by the total number of word tokens in the book. To divide, simply use the forward slash character.<sup>23</sup>

```
# Put a count of the occurrences of whale into whale.hits.v
whale.hits.v <- length(moby.word.v[which(moby.word.v=="whale")])

# Put a count of total words into total.words.v
total.words.v <- length(moby.word.v)

# now divide
whale.hits.v/total.words.v
## [1] 0.0053516
```

More interesting, perhaps, is to have R calculate the number of unique word types in the novel. R's *unique* function will examine all the values in the character vector and identify those that are the same and those that are different. By combining the *unique* and *length* functions, you calculate the number of unique words in Melville's *Moby Dick* vocabulary.

```
length(unique(moby.word.v))
## [1] 16872
```

<sup>22</sup> In R, as in many languages, there are often alternative ways of achieving the same goal. A more *elegant* method for calculating the number of *whale* hits might look like this: `length(moby.word.v[moby.word.v=="whale"])`. For beginners, the explicit use of *which* can be easier to understand.

<sup>23</sup> In these next few lines of code, I have added some comments to explain what the code is doing. This is a good habit for you to adopt; explaining or *commenting* your code so that you and others will be able to understand it later on. In R you insert comments into code by using a *#* symbol before the comment. When processing your code, R ignores everything between that *#* and the next full line return.

It turns out that Melville has a fairly big vocabulary: In *Moby Dick* Melville uses 16,872 unique words. That's interesting, but let's kick it up another notch. What we really want to know is how often he uses each of his words and which words are his favorites. We may even want to see if *Moby Dick* abides by Zipf's law regarding the general frequency of words in English.<sup>24</sup> No problem. R's `table` function can be used to build a "contingency" table of word types and their corresponding frequencies.

```
moby.freqs.t <- table(moby.word.v)
```

You can view the first few using `moby.freqs.t[1:10]`, and the entire frequency table can be sorted from most frequent to least frequent words using the `sort` function like this:

```
sorted.moby.freqs.t <- sort(moby.freqs.t , decreasing=TRUE)
```

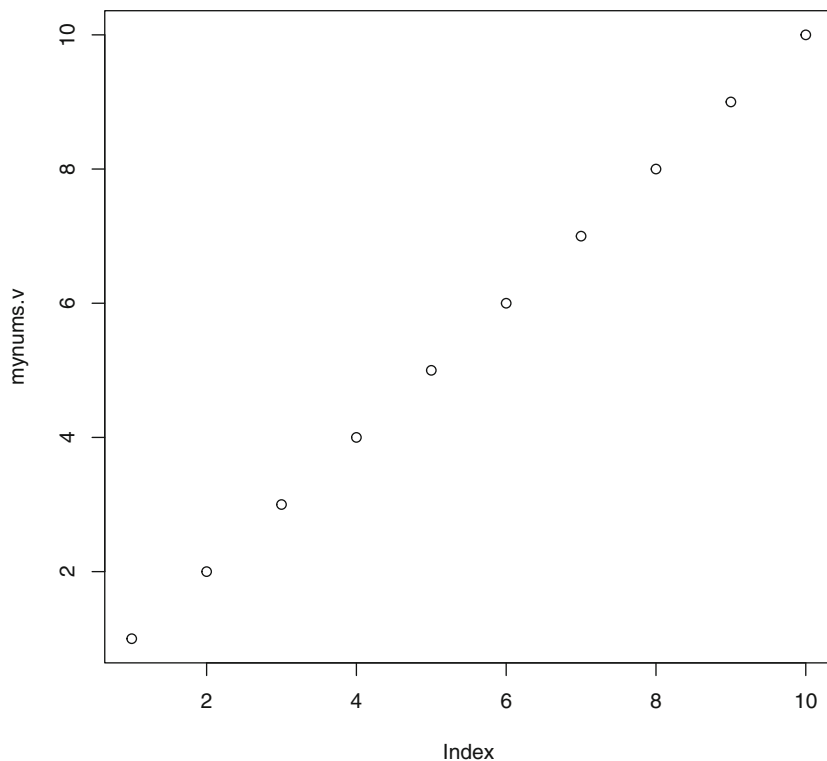
## Practice

**2.1.** Having sorted the frequency data as described in Sect. 2.4, figure out how to get a list of the top ten most frequent words in the novel. If you need a hint, go back and look at how we found the words "call" "me" "ishmael" in `moby.word.v`. Once you have the words, use R's built in `plot` function to visualize whether the frequencies of the words correspond to Zipf's law. The `plot` function is fairly straightforward. To learn more about the `plot`'s complex arguments, just enter: `?plot`. To complete this exercise, consider this example:

```
mynums.v <- c(1:10)  
plot(mynums.v)
```

---

<sup>24</sup> According to Zipf's law, the frequency of any word in a corpus is inversely proportional to its "rank" or position in the overall frequency distribution. In other words, the second most frequent word will occur about half as often as the most frequent word.



You only need to substitute the `mynums.v` variable with the top ten values from `sorted.moby.freqs.t`. You do not need to enter them manually!<sup>25</sup>

---

<sup>25</sup> When generating plots in `Rstudio`, you may get an error saying: “Error in `plot.new()` : figure margins too large.” This is because you have not given enough *screen real estate* to the *plots* pane of `Rstudio`. You can click and drag the frames of the plotting pane to resolve this issue.

Text Analysis with R for Students of Literature

Jockers, M.L.

2014, XVI, 194 p. 40 illus., 10 illus. in color., Hardcover

ISBN: 978-3-319-03163-7