7.8: Working with Text

Sometimes your data set is quite text heavy. This can be for a lot of different reasons. Maybe the raw data are actually taken from text sources (e.g., newspaper articles), or maybe your data set contains a lot of free responses to survey questions, in which people can write whatever text they like in response to some query. Or maybe you just need to rejig some of the text used to describe nominal scale variables. Regardless of what the reason is, you'll probably want to know a little bit about how to handle text in R. Some things you already know how to do: I've discussed the use of `nchar()` to calculate the number of characters in a string (Section 3.8.1), and a lot of the general purpose tools that I've discussed elsewhere (e.g., the `==` operator) have been applied to text data as well as to numeric data. However, because text data is quite rich, and generally not as well structured as numeric data, R provides a lot of additional tools that are quite specific to text. In this section I discuss only those tools that come as part of the base packages, but there are other possibilities out there: the `stringr` package provides a powerful alternative that is a lot more coherent than the basic tools, and is well worth looking into.

7.8.1 Shortening a string

The first task I want to talk about is how to shorten a character string. For example, suppose that I have a vector that contains the names of several different animals:

```
animals <- c( "cat", "dog", "kangaroo", "whale" )
```

It might be useful in some contexts to extract the first three letters of each word. This is often useful when annotating figures, or when creating variable labels: it's often very inconvenient to use the full name, so you want to shorten it to a short code for space reasons. The `strtrim()` function can be used for this purpose. It has two arguments: `x` is a vector containing the text to be shortened and `width` specifies the number of characters to keep. When applied to the `animals` data, here's what we get:

```
strtrim( x = animals, width = 3 )
```

```
## [1] "cat" "dog" "kan" "wha"
```

Note that the only thing that `strtrim()` does is chop off excess characters at the end of a string. It doesn't insert any whitespace characters to fill them out if the original string is shorter than the `width` argument. For example, if I trim the `animals` data to 4 characters, here's what I get:

```
strtrim( x = animals, width = 4 )
```

```
## [1] "cat"  "dog"  "kang" "whal"
```

The `"cat"` and `"dog"` strings still only use 3 characters. Okay, but what if you don't want to start from the first letter? Suppose, for instance, I only wanted to keep the second and third letter of each word. That doesn't happen quite as often, but there are some situations where you need to do something like that. If that does happen, then the function you need is `substr()`, in which you specify a `start` point and a `stop` point instead of specifying the width. For instance, to keep only the 2nd and 3rd letters of the various `animals`, I can do the following:

```
substr( x = animals, start = 2, stop = 3 )
```

```
## [1] "at" "og" "an" "ha"
```

7.8.2 Pasting strings together

Much more commonly, you will need either to glue several character strings together or to pull them apart. To glue several strings together, the `paste()` function is very useful. There are three arguments to the `paste()` function:

- `...` As usual, the dots "match" up against any number of inputs. In this case, the inputs should be the various different strings you want to paste together.
- `sep` . This argument should be a string, indicating what characters R should use as separators, in order to keep each of the original strings separate from each other in the pasted output. By default the value is a single space, `sep = " "` . This is made a little clearer when we look at the examples.
- `collapse` . This is an argument indicating whether the `paste()` function should interpret vector inputs as things to be collapsed, or whether a vector of inputs should be converted into a vector of outputs. The default value is `collapse = NULL` which is interpreted as meaning that vectors should not be collapsed. If you want to collapse vectors into as single string, then you should specify a value for `collapse` . Specifically, the value of `collapse` should correspond to the separator character that you want to use for the collapsed inputs. Again, see the examples below for more details.

That probably doesn't make much sense yet, so let's start with a simple example. First, let's try to paste two words together, like this:

```
paste( "hello", "world" )
```

```
## [1] "hello world"
```

Notice that R has inserted a space between the `"hello"` and `"world"` . Suppose that's not what I wanted. Instead, I might want to use `.` as the separator character, or to use no separator at all. To do either of those, I would need to specify `sep = "."` or `sep = ""` .[121] For instance:

```
paste( "hello", "world", sep = "." )
```

```
## [1] "hello.world"
```

Now let's consider a slightly more complicated example. Suppose I have two vectors that I want to `paste()` together. Let's say something like this:

```
hw <- c( "hello", "world" )
ng <- c( "nasty", "government" )
```

And suppose I want to paste these together. However, if you think about it, this statement is kind of ambiguous. It could mean that I want to do an "element wise" paste, in which all of the first elements get pasted together (`"hello nasty"`) and all the second elements get pasted together (`"world government"`). Or, alternatively, I might intend to collapse everything into one big string (`"hello nasty world government"`). By default, the `paste()` function assumes that you want to do an element-wise paste:

```
paste( hw, ng )
```

```
## [1] "hello nasty"      "world government"
```

However, there's nothing stopping you from overriding this default. All you have to do is specify a value for the `collapse` argument, and R will chuck everything into one dirty big string. To give you a sense of exactly how this works, what I'll do in this next example is specify *different* values for `sep` and `collapse` :

```
paste( hw, ng, sep = ".", collapse = ":::")
```

```
## [1] "hello.nasty:::world.government"
```

7.8.3 Splitting strings

At other times you have the opposite problem to the one in the last section: you have a whole lot of text bundled together into a single string that needs to be pulled apart and stored as several different variables. For instance, the data set that you get sent might include a single variable containing someone's full name, and you need to separate it into first names and last names. To do this in R you can use the `strsplit()` function, and for the sake of argument, let's assume that the string you want to split up is the following string:

```
monkey <- "It was the best of times. It was the blurst of times."
```

To use the `strsplit()` function to break this apart, there are three arguments that you need to pay particular attention to:

- `x` . A vector of character strings containing the data that you want to split.
- `split` . Depending on the value of the `fixed` argument, this is either a fixed string that specifies a delimiter, or a regular expression that matches against one or more possible delimiters. If you don't know what regular expressions are (probably most readers of this book), don't use this option. Just specify a separator string, just like you would for the `paste()` function.
- `fixed` . Set `fixed = TRUE` if you want to use a fixed delimiter. As noted above, unless you understand regular expressions this is definitely what you want. However, the default value is `fixed = FALSE` , so you have to set it explicitly.

Let's look at a simple example:

```
monkey.1 <- strsplit( x = monkey, split = " ", fixed = TRUE )
monkey.1
```

```
## [[1]]
##  [1] "It"     "was"    "the"    "best"   "of"     "times." "It"
##  [8] "was"    "the"    "blurst" "of"     "times."
```

One thing to note in passing is that the output here is a list (you can tell from the part of the output), whose first and only element is a character vector. This is useful in a lot of ways, since it means that you can input a character vector for `x` and then then have the `strsplit()` function split all of them, but it's kind of annoying when you only have a single input. To that end, it's useful to know that you can `unlist()` the output:

```
unlist( monkey.1 )
```

```
##  [1] "It"     "was"    "the"    "best"   "of"     "times." "It"
##  [8] "was"    "the"    "blurst" "of"     "times."
```

To understand why it's important to remember to use the `fixed = TRUE` argument, suppose we wanted to split this into two separate sentences. That is, we want to use `split = "."` as our delimiter string. As long as we tell R to remember to treat this as a *fixed* separator character, then we get the right answer:

```
strsplit( x = monkey, split = ".", fixed = TRUE )
```

```
## [[1]]
## [1] "It was the best of times"    " It was the blurst of times"
```

However, if we don't do this, then R will assume that when you typed `split = "."` you were trying to construct a "regular expression", and as it happens the character `.` has a special meaning within a regular expression. As a consequence, if you forget to include the `fixed = TRUE` part, you won't get the answers you're looking for.

7.8.4 Making simple conversions

A slightly different task that comes up quite often is making transformations to text. A simple example of this would be converting text to lower case or upper case, which you can do using the `toupper()` and `tolower()` functions. Both of these functions have a single argument `x` which contains the text that needs to be converted. An example of this is shown below:

```
text <- c( "lIfe", "Impact" )
tolower( x = text )
```

```
## [1] "life"   "impact"
```

A slightly more powerful way of doing text transformations is to use the `chartr()` function, which allows you to specify a "character by character" substitution. This function contains three arguments, `old`, `new` and `x`. As usual `x` specifies the text that needs to be transformed. The `old` and `new` arguments are strings of the same length, and they specify how `x` is to be converted. Every instance of the first character in `old` is converted to the first character in `new` and so on. For instance, suppose I wanted to convert `"albino"` to `"libido"`. To do this, I need to convert all of the `"a"` characters (all 1 of them) in `"albino"` into `"l"` characters (i.e., `a` → `l`). Additionally, I need to make the substitutions `l` → `i` and `n` → `d`. To do so, I would use the following command:

```
old.text <- "albino"
chartr( old = "aln", new = "lid", x = old.text )
```

```
## [1] "libido"
```

7.8.5 Applying logical operations to text

In Section 3.9.5 we discussed a very basic text processing tool, namely the ability to use the equality operator `==` to test to see if two strings are identical to each other. However, you can also use other logical operators too. For instance R also allows you to use the `<` and `>` operators to determine which of two strings comes first, alphabetically speaking. Sort of. Actually, it's a bit more complicated than that, but let's start with a simple example:

```
"cat" < "dog"
```

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

In this case, we see that `"cat"` does does come before `"dog"` alphabetically, so R judges the statement to be true. However, if we ask R to tell us if `"cat"` comes before `"anteater"`,

```
"cat" < "anteater"
```

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

It tell us that the statement is false. So far, so good. But text data is a bit more complicated than the dictionary suggests. What about `"cat"` and `"CAT"`? Which of these comes first? Let's try it and find out:

```
"CAT" < "cat"
```

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

In other words, R assumes that uppercase letters come before lowercase ones. Fair enough. No-one is likely to be surprised by that. What you might find surprising is that R assumes that *all* uppercase letters come before *all* lowercase ones. That is, while `"anteater" < "zebra"` is a true statement, and the uppercase equivalent `"ANTEATER" < "ZEBRA"` is also true, it is *not* true to say that `"anteater" < "ZEBRA"`, as the following extract illustrates:

```
"anteater" < "ZEBRA"
```

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

This may seem slightly counterintuitive. With that in mind, it may help to have a quick look Table 7.3, which lists various text characters in the order that R uses.

Table 7.3: The ordering of various text characters used by the < and > operators, as well as by the sort() function. Not shown is the "space" character, which actually comes rst on the list.

Characters
! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ? @ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [ ] ^ _ ‘ a b c d e f g h i j k l m n o p q r s t u v w x y z } | {

One function that I want to make a point of talking about, even though it's not quite on topic, is the `cat()` function. The `cat()` function is a of mixture of `paste()` and `print()`. That is, what it does is concatenate strings and then print them out. In your own work you can probably survive without it, since `print()` and `paste()` will actually do what you need, but the `cat()` function is so widely used that I think it's a good idea to talk about it here. The basic idea behind `cat()` is straightforward. Like `paste()`, it takes several arguments as inputs, which it converts to strings, collapses (using a separator character specified using the `sep` argument), and prints on screen. If you want, you can use the `file` argument to tell R to print the output into a file rather than on screen (I won't do that here). However, it's important to note that the `cat()` function collapses vectors first, and *then* concatenates them. That is, notice that when I use `cat()` to combine `hw` and `ng`, I get a different result than if I'd used `paste()`

```
cat( hw, ng )
```

```
## hello world nasty government
```

```
paste( hw, ng, collapse = " " )
```

```
## [1] "hello nasty world government"
```

Notice the difference in the ordering of words. There's a few additional details that I need to mention about `cat()`. Firstly, `cat()` really is a function for *printing*, and not for creating text strings to store for later. You can't assign the output to a variable, as the following example illustrates:

```
x <- cat( hw, ng )
```

```
## hello world nasty government
```

```
x
```

```
## NULL
```

Despite my attempt to store the output as a variable, `cat()` printed the results on screen anyway, and it turns out that the variable I created doesn't contain anything at all.[122] Secondly, the `cat()` function makes use of a number of "special" characters. I'll talk more about these in the next section, but I'll illustrate the basic point now, using the example of `"\n"` which is interpreted as a "new line" character. For instance, compare the behaviour of `print()` and `cat()` when asked to print the string `"hello\nworld"` :

```
print( "hello\nworld" )  # print literally:
```

```
## [1] "hello\nworld"
```

```
cat( "hello\nworld" )  # interpret as newline
```

```
## hello
## world
```

In fact, this behaviour is important enough that it deserves a section of its very own…

7.8.6 Using escape characters in text

The previous section brings us quite naturally to a fairly fundamental issue when dealing with strings, namely the issue of delimiters and escape characters. Reduced to its most basic form, the problem we have is that R commands are written using text characters, and our strings also consist of text characters. So, suppose I want to type in the word "hello", and have R encode it as a string. If I were to just type `hello` , R will think that I'm referring to a variable or a function called `hello` rather than interpret it as a string. The solution that R adopts is to require you to enclose your string by ***delimiter*** characters, which can be either double quotes or single quotes. So, when I type `"hello"` or `'hello'` then R knows that it should treat the text in between the quote marks as a character string. However, this isn't a complete solution to the problem: after all, `"` and `'` are themselves perfectly legitimate text characters, and so we might want to include those in our string as well. For instance, suppose I wanted to encode the name "O'Rourke" as a string. It's *not* legitimate for me to type `'O'rourke'` because R is too stupid to realise that "O'Rourke" is a real word. So it will interpret the `'O'` part as a complete string, and then will get confused when it reaches the `Rourke'` part. As a consequence, what you get is an error message:

```
'O'Rourke'
Error: unexpected symbol in "'O'Rourke"
```

To some extent, R offers us a cheap fix to the problem because of the fact that it allows us to use either `"` or `'` as the delimiter character. Although `'O'rourke'` will make R cry, it is perfectly happy with `"O'Rourke"` :

```
"O'Rourke"
```

```
## [1] "O'Rourke"
```

This is a real advantage to having two different delimiter characters. Unfortunately, anyone with even the slightest bit of deviousness to them can see the problem with this. Suppose I'm reading a book that contains the following passage,

P.J. O'Rourke says, "Yay, money!". It's a joke, but no-one laughs.

and I want to enter this as a string. Neither the `'` or `"` delimiters will solve the problem here, since this string contains both a single quote character and a double quote character. To encode strings like this one, we have to do something a little bit clever.

Table 7.4: Standard escape characters that are evaluated by some text processing commands, including `cat()` . This convention dates back to the development of the C programming language in the 1970s, and as a consequence a lot of these characters make

most sense if you pretend that R is actually a typewriter, as explained in the main text. Type `?Quotes` for the corresponding R help file.

Escape.sequence	Interpretation
`\n`	Newline
`\t`	Horizontal Tab
`\v`	Vertical Tab
`\b`	Backspace
`\r`	Carriage Return
`\f`	Form feed
`\a`	Alert sound
`\\`	Backslash
`\'`	Single quote
`\"`	Double quote

The solution to the problem is to designate an ***escape character***, which in this case is `\`, the humble backslash. The escape character is a bit of a sacrificial lamb: if you include a backslash character in your string, R will *not* treat it as a literal character at all. It's actually used as a way of inserting "special" characters into your string. For instance, if you want to force R to insert actual quote marks into the string, then what you actually type is `\'` or `\"` (these are called ***escape sequences***). So, in order to encode the string discussed earlier, here's a command I could use:

```
PJ <- "P.J. O\'Rourke says, \"Yay, money!\". It\'s a joke, but no-one laughs."
```

Notice that I've included the backslashes for both the single quotes and double quotes. That's actually overkill: since I've used `"` as my delimiter, I only needed to do this for the double quotes. Nevertheless, the command has worked, since I didn't get an error message. Now let's see what happens when I print it out:

```
print( PJ )
```

```
## [1] "P.J. O'Rourke says, \"Yay, money!\". It's a joke, but no-one laughs."
```

Hm. Why has R printed out the string using `\"`? For the exact same reason that *I* needed to insert the backslash in the first place. That is, when R prints out the `PJ` string, it has enclosed it with delimiter characters, and it wants to unambiguously show us which of the double quotes are delimiters and which ones are actually part of the string. Fortunately, if this bugs you, you can make it go away by using the `print.noquote()` function, which will just print out the literal string that you encoded in the first place:

```
print.noquote( PJ )
```

Typing `cat(PJ)` will produce a similar output.

Introducing the escape character solves a lot of problems, since it provides a mechanism by which we can insert all sorts of characters that aren't on the keyboard. For instance, as far as a computer is concerned, "new line" is actually a text character. It's the character that is printed whenever you hit the "return" key on your keyboard. If you want to insert a new line character into your string, you can actually do this by including the escape sequence `\n`. Or, if you want to insert a backslash character, then you can use `\\`. A list of the standard escape sequences recognised by R is shown in Table 7.4. A lot of these actually date back to the days of the typewriter (e.g., carriage return), so they might seem a bit counterintuitive to people who've never used one. In

order to get a sense for what the various escape sequences do, we'll have to use the `cat()` function, because it's the only function "dumb" enough to literally print them out:

```
cat( "xxxx\boo" )  # \b is a backspace, so it deletes the preceding x
cat( "xxxx\too" )  # \t is a tab, so it inserts a tab space
cat( "xxxx\noo" )  # \n is a newline character
cat( "xxxx\roo" )  # \r returns you to the beginning of the line
```

And that's pretty much it. There are a few other escape sequence that R recognises, which you can use to insert arbitrary ASCII or Unicode characters into your string (type `?Quotes` for more details) but I won't go into details here.

7.8.7 Matching and substituting text

Another task that we often want to solve is find all strings that match a certain criterion, and possibly even to make alterations to the text on that basis. There are several functions in R that allow you to do this, three of which I'll talk about briefly here: `grep()`, `gsub()` and `sub()`. Much like the `substr()` function that I talked about earlier, all three of these functions are intended to be used in conjunction with regular expressions (see Section 7.8.9 but you can also use them in a simpler fashion, since they all allow you to set `fixed = TRUE`, which means we can ignore all this regular expression rubbish and just use simple text matching.

So, how do these functions work? Let's start with the `grep()` function. The purpose of this function is to input a vector of character strings `x`, and to extract all those strings that fit a certain pattern. In our examples, I'll assume that the `pattern` in question is a literal sequence of characters that the string must contain (that's what `fixed = TRUE` does). To illustrate this, let's start with a simple data set, a vector that contains the names of three `beers`. Something like this:

```
beers <- c( "little creatures", "sierra nevada", "coopers pale" )
```

Next, let's use `grep()` to find out which of these strings contains the substring `"er"`. That is, the `pattern` that we need to match is the fixed string `"er"`, so the command we need to use is:

```
grep( pattern = "er", x = beers, fixed = TRUE )
```

```
## [1] 2 3
```

What the output here is telling us is that the second and third elements of `beers` both contain the substring `"er"`. Alternatively, however, we might prefer it if `grep()` returned the actual strings themselves. We can do this by specifying `value = TRUE` in our function call. That is, we'd use a command like this:

```
grep( pattern = "er", x = beers, fixed = TRUE, value = TRUE )
```

```
## [1] "sierra nevada" "coopers pale"
```

The other two functions that I wanted to mention in this section are `gsub()` and `sub()`. These are both similar in spirit to `grep()` insofar as what they do is search through the input strings (`x`) and find all of the strings that match a `pattern`. However, what these two functions do is *replace* the pattern with a `replacement` string. The `gsub()` function will replace *all* instances of the pattern, whereas the `sub()` function just replaces the first instance of it in each string. To illustrate how this works, suppose I want to replace all instances of the letter `"a"` with the string `"BLAH"`. I can do this to the `beers` data using the `gsub()` function:

```
gsub( pattern = "a", replacement = "BLAH", x = beers, fixed = TRUE )
```

```
## [1] "little creBLAHtures"    "sierrBLAH nevBLAHdBLAH"
## [3] "coopers pBLAHle"
```

Notice that all three of the `"a"` s in `"sierra nevada"` have been replaced. In contrast, let's see what happens when we use the exact same command, but this time using the `sub()` function instead:

```
sub( pattern = "a", replacement = "BLAH", x = beers, fixed = TRUE )
```

```
## [1] "little creBLAHtures" "sierrBLAH nevada"    "coopers pBLAHle"
```

Only the first `"a"` is changed.

7.8.8 Regular expressions (not really)

There's one last thing I want to talk about regarding text manipulation, and that's the concept of a ***regular expression***. Throughout this section we've often needed to specify `fixed = TRUE` in order to force R to treat some of our strings as actual strings, rather than as regular expressions. So, before moving on, I want to very briefly explain what regular expressions are. I'm *not* going to talk at all about how they work or how you specify them, because they're genuinely complicated and not at all relevant to this book. However, they are extremely powerful tools and they're quite widely used by people who have to work with lots of text data (e.g., people who work with natural language data), and so it's handy to at least have a vague idea about what they are. The basic idea is quite simple. Suppose I want to extract all strings in my `beers` vector that contain a vowel followed immediately by the letter `"s"`. That is, I want to finds the beer names that contain either `"as"`, `"es"`, `"is"`, `"os"` or `"us"`. One possibility would be to manually specify all of these possibilities and then match against these as fixed strings one at a time, but that's tedious. The alternative is to try to write out a single "regular" expression that matches all of these. The regular expression that does this[123] is `"[aeiou]s"`, and you can kind of see what the syntax is doing here. The bracketed expression means "any of the things in the middle", so the expression as a whole means "any of the things in the middle" (i.e. vowels) followed by the letter `"s"`. When applied to our beer names we get this:

```
grep( pattern = "[aeiou]s", x = beers, value = TRUE )
```

```
## [1] "little creatures"
```

So it turns out that only `"little creatures"` contains a vowel followed by the letter `"s"`. But of course, had the data contained a beer like `"fosters"`, that would have matched as well because it contains the string `"os"`. However, I deliberately chose not to include it because Fosters is not – in my opinion – a proper beer.[124] As you can tell from this example, regular expressions are a neat tool for specifying *patterns* in text: in this case, "vowel then s". So they are definitely things worth knowing about if you ever find yourself needing to work with a large body of text. However, since they are fairly complex and not necessary for any of the applications discussed in this book, I won't talk about them any further.
