Chapter 10
Do It KWIC(er) (and Better)

Abstract This chapter expands upon the previous chapter in order to build an interactive and reusable Keyword in Context (KWIC) application that allows for quick and intuitive KWIC list building. Readers are introduced to interactive R functions including `readline` and functions for data type conversion.

10.1 Getting Organized

In the previous chapter, you learned how to find and access a series of index positions in a vector and then how to return values on either side of the found positions. In the practice exercise, you hard-coded a solution for finding occurrences of the word *dog* in *Sense and Sensibility* and *Moby Dick*. In this section you will learn how to abstract that code and how to create an interactive and reusable application that will allow you to repeatedly find keywords in context without having to hard-code the search terms.

If you have not already done so, now is the time to get organized. You will be dealing with more and more files as this book continues, and unless you keep your working spaces well-defined and organized things can get complicated. Within your "TAWR2" directory, you already have a sub-directory labeled "code." This is where you should be storing all of your .R files. Now is a good time to create a new sub-directory called "results." In the last exercise in this chapter, you will be generating a .csv file that you can save in your "results" directory and then open again in R or in a spreadsheet application such as Excel or Open Office.

M. L. Jockers, R. Thalken, *Text Analysis with R*, Quantitative Methods in the Humanities and Social Sciences,
https://doi.org/10.1007/978-3-030-39643-5_10

10.2 Separating Functions for Reuse

In the last chapter you created two functions, and in this chapter you will create a third. Because you can reuse functions in separate projects, it is convenient to keep them in a separate file so that you can access them from different R scripts that you write for different projects. You should begin this chapter, therefore, by copying your two functions from the last chapter into a new file that you will title *corpus_functions.R*. Save this new file inside your "code" sub-directory. Your functions file should include both `show_files` and `make_token_v` from the last chapter. Here they are again, but without the comments:

```
show_files  <- function(directory_path, pattern = "\\.txt$"){
  file_name_v <- dir(directory_path, pattern, full.names = TRUE)
  for(i in seq_along(file_name_v)){
    cat(i, file_name_v[i], "\n", sep = " ")
  }
}
make_token_v <- function(file_path, pattern = "\\W"){
  text_v <- scan(file_path, what = "character", sep = "\n")
  text_v <- paste(text_v, collapse = " ")
  text_lower_v  <- tolower(text_v)
  text_words_v <- strsplit(text_lower_v, pattern)
  text_words_v <- unlist(text_words_v)
  text_words_v[which(text_words_v != "")]
}
```

With your functions stored in a separate file, you can now *call* the *corpus_functions.R* file as part of your working R script in order to load these existing functions. Create a new R script (saved as "chapter10.R" in your "code" directory) and enter the following expressions as the first two lines:

```
rm(list = ls())
source("code/corpus_functions.R")
```

The first line clears your workspace and the second line uses R's `source` function to load the contents of your external functions file. When this script is executed, R will load all of the functions that you create and save in the *corpus_functions.R* file.

As in Chap. 9, you need to show R where to find your text files, so next you will define an *input directory* with a relative path to the `data/plainText` directory.

```
input_dir <- "data/text"
```

Since you also will be using R to create derivative data files that will need to be saved out to another directory, you will need to tell R where to write these files. Define an *output directory* variable, with the title "results," like this:

```
output_dir  <- "results"
```

The objective now is to write an interactive Keyword in Context (KWIC) function that will allow you to repeatedly enter different file paths and keywords and then return the hits for those terms along with some amount of context on either side of the key term.

10.3 User Interaction

R includes a set of built-in functions that, when invoked, require user feedback. Thus far we have been hard-coding file paths in R, but we could have been using R's `file.choose` function instead. If you enter `file.choose` at the R prompt, you will be prompted with a pop-up window that allows you to navigate your file system and locate a file. Here is an example that you can try on your system. Just enter the following expression at the R prompt in the console pane and then use your computer's windowing system to locate the file in the exercise directory called "melville.txt."

```
mytext <- scan(file.choose(), what = "character", sep = "\n")
```

If you did everything correctly, you should see the message:

```
Read 18172 items
```

You will now be able to enter

```
mytext
```

and see all the lines of *Moby Dick*.

10.4 readline

There are other functions in R that allow for user interaction as well, and one that we will use for this section is `readline`. `readline` is a function that will print information to the R console and then accept input entered into the console by the user. Enter this expression into the console and hit return:

```
myyear <- readline("What year was Moby Dick published? \n")
```

You will see the quoted question appear in the console and the blinking cursor prompt located after the question mark. At the cursor prompt, enter a number (e.g., `1851`) and hit return. If you now type `myyear` at the R prompt and hit return, you will find that R has stored the value that you entered in the `myyear` variable. Here is how it should look:

```
> myyear <- readline("What year was Moby Dick published? \n")
What year was Moby Dick published? 1851
> myyear
[1] "1851"
```

10.5 Building a Better KWIC Function

Using the `readline` function, you can write a *KWIC* list function that asks the user (you) for a *file* to search, a *keyword* to find, and an amount of *context* to be returned on either side of the keyword. We will name this function `doitKwic` and call it in this fashion:

```
doitKwic(directory_path)
```

The only argument that you need to send this function is the location of (path to) a directory on your system. Open your *corpus_functions.R* file and begin writing this new function like this:

```
doitKwic <- function(directory_path){
  # instructions here will ask user for a file to search
  # a keyword to find and a "context" number for context
  # on either side of the of the keyword
}
```

Keep in mind that the argument name used inside the parentheses of the function does not have to be the same as the name used outside of the function. You already have an object called `input_dir` instantiated from above. This object contains the path expression `"data/text"` that is the location of two plain text files. So here we are defining a function that takes an argument called `directory_path`, and when we call this function, we will send it the information contained in the `input_dir` object.

You do not have to write your code this way (i.e., using different names when inside or outside of the function), but we find it useful to name our function arguments in a way that is descriptive of their content and a bit more abstract than the names we give to objects within the main script. We may decide to

use this function on another project, and several months from now we may have forgotten what `input_dir` means. Using `directory_path` is a bit more descriptive, and it gives us some clues about what kind of data the function is expecting.

As the commented sections of the code suggest, we want the new function to ask the user for input. First it needs to ask which file in the directory to search in, then what keyword to search for, and finally how much context to display. For the first item, the function should display a list of the files that are found inside the directory located at `directory_path` and then ask the user to choose one. As it happens, we already have a function called `show_files` that does exactly this, and we can call the `show_files` function from inside the new `doitKwic` function! Remember that `show_files` is expecting to get a directory path as its argument. That information is passed to `show_files` in the `directory_path` argument. So as a next step, we might write the following:

```
doitKwic <- function(directory_path){
  show_files(directory_path)
  # more instructions here . . .
}
```

If `doitKwic` is called, it will successfully show the files found in the directory sent as the argument `directory_path`, but then it will do nothing else. In order to capture information from the user, we will need to wrap the call to `show_files` inside a call to `readline`:

```
doitKwic <- function(directory_path){
  readline(show_files(directory_path))
  # more instructions here . . .
}
```

This gets us a little bit closer, but we are not there quite yet. Recall that `show_files` presents us with both an id number and a path for each file. When you call `show_files` using `data/text` you get the following output:

```
## 1 data/text/austen.txt
## 2 data/text/melville.txt
```

Instead of having to copy or type in the entire file path that we want to search in, let us have our user just enter the index number of the file instead. We will capture that user input into a new object called `file_id`.

```
doitKwic <- function(directory_path){
  file_id <- readline(show_files(directory_path))
  # more instructions here . . .
}
```

There is now one more thing we have to fix. The `readline` function accepts input as *character* data, so if the user enters the number `2`, to access the "melville.txt" file, that `2` is converted to the character "2." We must, therefore, convert, or recast, the character to a numeric value using `as.numeric`.

```
doitKwic <- function(directory_path){
  file_id <- as.numeric(readline(show_files(directory_path)))
  # more instructions here . . .
}
```

Now we can collect the other information we need: the keyword and the amount of context. We will add two more lines to our evolving function:

```
doitKwic <- function(directory_path){
  file_id <- as.numeric(readline(show_files(directory_path)))
  keyword <- readline("Enter a Keyword: ")
  context <- as.numeric(readline("How many words of context? "))
  # more instructions here . . .
}
```

Notice that we need to use `as.numeric` again in the last line to be sure the context the user enters is converted to a numeric value. With these three ingredients, we now have enough information to access, tokenize, and search for a keyword in a text file. The next thing to do is to take advantage of the function that we have already written for handling the tokenization: `make_token_v`. We will add a call to `make_token_v` to our function as follows:

```
doitKwic <- function(directory_path){
  file_id <- as.numeric(readline(show_files(directory_path)))
  keyword <- readline("Enter a Keyword: ")
  context <- as.numeric(readline("How many words of context? "))
  word_v <- make_token_v(
    dir(directory_path, full.names = TRUE)[file_id]
    )
  # more instructions here . . .
}
```

This last line is a bit complicated, so let us break it down. Recall that `make_token_v` takes a path argument. Here we have used the built-in `dir` function with the `full.names = TRUE` argument to return a file path using a combination of information that we have stored in the `directory_path` and `file_id` objects. Recall that calling `dir(directory_path, full.names = TRUE)` returns a vector object of file paths. We can access specific items in this vector using bracketed sub-setting, and the specific index of the item we want to access is now stored in the `file_id` object. Therefore, calling `dir(directory_path, full.names = TRUE)[file_id]` will return the precise path to a single file. That file is then sent to `make_token_v` where it is tokenized and returned into the `word_v` object.

All you need to do now is apply what you learned from the exercise in the last chapter. Using `which` you will identify the positions in the `word_v` object that match the user's keyword and store them in an object called `hits_v`. Then you will loop over the `hits_v` object using a `for` loop and along the way add and subtract the context values from the found positions in order to identify and display the user's keyword in context. The (almost) completed function looks like this:

```
doitKwic <- function(directory_path){
  file_id <- as.numeric(readline(show_files(directory_path)))
  keyword <- readline("Enter a Keyword: ")
  context <- as.numeric(readline("How many words of context? "))
  word_v <- make_token_v(
    file.path(directory_path, dir(directory_path)[file_id])
    )
  hits_v <- which(word_v == keyword)
  for(i in seq_along(hits_v)){
    start <- hits_v[i] - context
    end <- hits_v[i] + context
    before <- word_v[start:(start + context - 1)]
    after <- word_v[(start + context + 1):end]
    keyword <- word_v[start + context]
    cat("----------------------", i, "----------------------", "\n")
    cat(before,"[", keyword, "]", after, "\n")
  }
}
```

10.6 Fixing Some Problems

Unfortunately, this simple solution cannot handle all of the possible search scenarios that might occur, and we have left out some important arguments. Recall, for example, that by default, our `make_token_v` converts all characters to lowercase. If a user of our new `doitKwic` function were to enter a keyword containing a capital letter, nothing would be found. We can fix this very easily by altering the third line of the function to read `keyword <- tolower(readline("Enter a Keyword: "))`. This ensures that whatever the user enters will be converted to lowercase. But what if you *want* to search for a capitalized word? Right now that is not an option. And there is another more serious problem...

What if the very first word in the file you are searching in is a hit? In this case the first position in the `hits_v` vector would be `1` and that would cause `start` to be set to `1 - (minus) context`: that is one minus whatever number the user entered for `context`. The result of that subtraction would be a negative

number and R would choke trying to access a value held at a negative vector index! You cannot have that, so you need to add some code to deal with this possibility. Here is one way to deal with the problem using an `if` conditional:

```
start <- hits_v[i] - context
if(start < 1){
  start <- 1
}
```

A similar problem exists on the other end of the vector. What if the last word is a hit? Adding some amount of context after the last hit will result in R trying to return a value that does not exist after the last word. We can deal with this issue in a similar manner: if the value of `end` is greater than or equal to the length of the entire vector, we can set `end` equal to the length of the entire vector.

```
end <- hits_v[i] + context
if(end >= length(word_v)){
  end <- length(word_v)
}
```

We will deal with the lowercase issue and some other issues in the practice exercises, but for now we at least have a function that will not break. Here is the final version:

```
doitKwic <- function(directory_path){
  file_id <- as.numeric(readline(show_files(directory_path)))
  keyword <- readline("Enter a Keyword: ")
  context <- as.numeric(readline("How many words of context? "))
  word_v <- make_token_v(
    file.path(directory_path, dir(directory_path)[file_id])
    )
  hits_v <- which(word_v == keyword)
  for(i in seq_along(hits_v)){
    start <- hits_v[i] - context
    if(start < 1){
      start <- 1
    }
    end <- hits_v[i] + context
    if(end >= length(word_v)){
      end <- length(word_v)
    }
    output <- word_v[start:end]
    output[which(output == keyword)] <- paste(
      "[", keyword, "]", sep = ""
      )
```

```
    cat("----------------------", i, "----------------------", "\n")
    cat(output, "\n")
  }
}
```

Save this function to your *corpus_functions.R* file and then take it for a test run using the following code:

```
source("code/corpus_functions.R")
input_dir <- "data/text"
doitKwic(input_dir)
```

10.7 Practice

1. In prior exercises and lessons, you have learned how to instantiate an empty object outside of a `for` loop and then how to add new data to that object during the loop. You have learned how to use `cbind` to add columns of data and `rbind` to add rows. You have also learned how to use `paste` with the `collapse` argument to glue together pieces in a vector of values and how to use `cat` to concatenate items in a vector. And you have used `colnames` to get and set the names of columns in a data frame. Using all of this knowledge, modify the function written in this chapter (`doItKwic`) so that the results of a KWIC search are put into a `data frame` object in which each row is a single KWIC result. Name this new function `doItKwicBetter`. Your resulting data frame should have four columns labeled as follows: *position*, *left context*, *keyword*, and *right context*. The *position* column will contain the index value showing where in the file the keyword was found. The *left* column will contain the words in the file vector that were found to the left of the keyword. The *keyword* column will contain the keyword, and the *right* column, the context that was found to the right of the keyword. Here is an example of results generated using the keyword *dog* with two words of context in the file "melville.txt."

```
   position                      left keyword                right
## 1     10643       like a newfoundland    dog           just from the
## 2     12464              that in the     dog           days will mow
## 3     23280          swimming like a     dog       throwing his long
## 4     47119             at last down     dog     and kennel starting
## 5     47195              be called a     dog             sir then be
## 6     47653                call me a     dog       blazes he called
## 7     70018          the sacred white    dog              was by far
## 8    103702                lives in a    dog              or a horse
## 9    103788           kindness of the    dog      the accursed shark
## 10   133135 ungracious and ungrateful    dog       cried starbuck he
```

```
## 11   133165       give way greyhounds    dog                to it i
## 12   143092             whale that a     dog             does to the
## 13   163384        the ram lecherous     dog            he begets us
## 14   166285        bunger bunger you     dog           laugh out why
## 15   166665            in pickle you     dog           you should be
## 16   167192               and like a     dog strangely snuffing this
## 17   199028           feet high hang     dog       look and cowardly
## 18   202985         sagacious ship s     dog         will in drawing
## 19   203037              and then the    dog           vane and then
```

2. Copy the function you created in the exercise above and modify it to include a feedback loop asking the user if the results should be saved as a `.csv` file. If the user answers "y" for "yes," generate a file name based on the existing user input (keyword, file name, context) and write that file to the `results` directory using a call to the `write.csv` function, as in this example below. Save this new function in your *corpus_functions.R* file as `doItKwicStillBetter`.

```
write.csv(results_df, file.path("results", some_file_name))
```

3. Neither of these "better" KWIC functions gives the user any options for tokenizing the texts. Right now both functions rely on the default behavior of the `make_token_v` function, which uses the regular expression `"\\W"`. In order to give users flexibility to change the way files get tokenized, we need to alter the line of code that calls `make_token_v` to include a `pattern` argument, and we also need to add a new argument to the parameters of our KWIC function. Rewrite your `doItKwicStillBetter` function to achieve this objective and save it as `doItKwicBest`. After you save the function, you should be able to call it using the code shown below. Recall that `[^A-Za-z0-9']` is a regular expression that retains apostrophes and possessives. If your function is correct, you will be able to search for instances of *ahab's*. After you have coded this new version, check your solution with the solution at the back of the book where you will find one more useful iteration of this function explained. Once you have finished the practice exercises for this chapter, save `doItKwic`, `doItKwicBetter`, `doItKwicStillBetter`, and `doItKwicBest` to your *corpus_functions.R* file so you can easily access them in the future.

```
doItKwicBest(input_dir, "[^A-Za-z0-9']")
```