STAT243 - Problem Set 3

Cristobal Pais - cpaismz@berkeley.edu

September 29th, 2017

Problem 1

b) Selected Article

Selected text: “Best Practices for Scientific Computing, Wilson et. al”

c) Comments & Questions

i) In section 1 "Write programs for people, not computers", the authors state that the code style and formatting should be consistent giving an example where some variable names are following the CamelCaseNaming while others use pothole_case_naming fact that makes it more difficult (and longer) to read and readers make more mistakes.

In our opinion, this can be true under some assumptions but a generalization can be unfair: some programs (and programmers) use a mix of formats depending on the type of the variable, for example to separate global variables from local ones, numerical from strings, arrays, etc., or mutable parameters from immutable ones, making it easier for both the reader and other researchers to follow the logic of the script. Therefore, we do not completely agree with the authors in this point, although we agree with the other elements mentioned in the section.

- Is there a best practice rule or preferred style for accounting situations like the stated ones?
- Are there some differences depending on the chosen language?.
- Does the OS matter when using different formats/styles?

ii) The main point of section 2 "Automate repetitive tasks" is one of the most important but somewhat forgotten practices among scientists from different areas. Based on our experience, a large number of researchers end up making repetitive tasks by themselves (manually) instead of relying on the computer to repeat them. Situations like "I already make the main code but I do not want to automate this step, because I want more control" are pretty common among some researchers, however, they are actually doing the opposite: losing the complete control of the situation, adding potential errors and human mistakes.

- On the other hand, we did not know about the existence of software like Make.
- How do these software work?
- Which are the most popular ones?
- Are they being used in practice?
- Can we mix different languages and call them?

iii) The use of version control software is something that we believe is the most important tool when developing collaborative projects. Before taking the course, our classic “control” workflow was basically the use of some online (cloud) repository such as Dropbox, Google Drive and even a shared email for keeping all the relevant files, comments, log with the implemented changes of the current version, etc.However, it was clearly not the best option: we had a lot of problems with duplicated files, out-of-date replacements, lost files, a mix of old functions/classes with new ones, leading us to lose time, commit a lot of mistakes and in the end, the team was completely frustrated.

iv) When the authors indicate that researcher should “use a variety of oracles”, we are not completely sure about what they are having in mind:

- What is an example of a pertinent oracle?
- What happens if the researcher does not know what might be the "expected" output of the research? (like when using unsupervised machine learning algorithms) Should we use known experiments as a comparison/starting point?

v) Regarding the section “Optimize software only after it works correctly”, we do not completely agree with the authors, since depending on what your project/function/program aims, you should have a good idea about which language(s) is/are most suitable for that implementation, otherwise, depending on the "first language" you selected in order to have a “fast development” (like with a high-level language) the team could end up working maybe twice or more than the original estimated amount of hours allocated for that particular step, thus incurring in a series of costs in terms of time and monetary evaluation.

Therefore, we believe that, although it is a good practice/idea to get a prototype fast, it cannot be done without thinking in advance that we might need to translate/modify the original code into a new language in order to obtain the performance needed by the project. For example, we think that is easier to translate a piece of python/java code into **C** than from **R**. Hence, we do not agree with the sentence “write code in the highest-level language possible” mentioned by the authors: we should write the code in the language that is appropriate for the project and that the team feels comfortable to work with.

- Is there some sort of a “guide of compatibility’ or “easy-to-translate” document that summarizes the best practices for porting the code from one language to another?
- Which languages are "good friends"? (easier to port/translate from one format to the other).
- Are there formal studies/articles about this topic?.
- Is there a preferred language today in research or still depends on the project itself?
- Which languages are being used in large computer labs like Sandia, Nasa, etc.?

vi) The pair programming collaborative strategy does not look like a useful approach in our opinion. Programmers usually have their own working style, say: code structure, working environment (including details such as the type of music they want to listen while coding), rhythm (e.g. when do they want to have a break), etc. Therefore, the practice seems very intrusive and poor, with a very high probability of being disruptive instead of productive.

In our opinion, is far better to discuss the problem/function to be coded before starting coding such that all the members of the team who are actually involved in that piece of the project are “on the same page” and know exactly what they need to do, how they are going to tackle it and who is the responsible for what. After every progress step, the team can have a brief meeting, sharing all the relevant aspects of the current version, current progress and the next goal/aims to reach in the next step.

- Is this practice (pair programming) actually being used in companies/research teams?
- How are the studies measuring the supposed “improvement in productivity” reached by the use of this practice?
- Would you recommend to apply this under some particular context?

Problem 2

The general strategy to solve this problem consists of performing a smart pre-processing of the original text, taking into account its particular structure/format for the different components of each play such as the text indentation, the new lines location, the presence of special characters/keywords/chunks included by default for separating the plays (such as the end of each play, title and year, acts and scene markers, etc.), and the identification of special cases that must be treated with a specific logic in order to be successfully processed by a set of regular expressions without affecting other components of the play (like the difficult to deal "Songs" chunks). Thus, based on our knowledge of the file structure, we define a detailed set of different regular expressions that will allow us to extract/modify the contents of the original .txt file. All the relevant expressions are used inside a *Preprocessing()* script that contains two functions that will pre-process the text file in two main steps.

As a first pre processing step, all the beginning of the file and the last piece (Lover's Complaint) are deleted from the file based on the lines associated with them inside the txt file, following the instructions from the problem statement. Then, thanks to our two pre processing functions *PreProcessing()* and *PreProcessing2()*, we are able to insert a series of special characters inside the original .txt file identifying a large number of scene directions, the main speakers of each play, scenes and acts information, title of the play, songs, epilogues, prologues and special elements (instructions/directions or speaker names that do not follow any of the previously identified patterns). In addition, an initial cleaning procedure is performed, taking out all the copyright text in between << and >> special characters.

Once we have the already pre-processed text file/object we can easily split it by plays (36 of them) using the fact that a special line including "THE END" string, obtaining a list of 36 elements, each one representing a full pre-processed play. Therefore, the pre-processing step is the most relevant and important step of our solution approach since it will allow us to easily perform the rest of the string manipulations needed for extracting the relevant components from each play.

Thanks to our significant pre-processing step, we can easily extract all the metadata contained in each play by exploiting the file structure. Using the special characters inserted inside each play, we are able to declare short and explicit regular expressions that will help us to extract the title of the play, the year of publication, the number of acts, and the number of scenes. Important is to note that we developed a series of short and easy to use functions (see details below) in order to take advantage of repetitive tasks, improving the code efficiency, "easy-to-follow" structure and modularity using both vectorized operations and loops using the well-known *lapply()* function.

Then, a final cleaning step for both normal and special cases is performed. All the extra elements such as stage directions, dramatis personae information, and scene information are cleaned from the main text object in order to extract all the speakers and their corresponding spoken text chunks. Again, we exploit our pre-processing structure in order to easily eliminate the majority of this elements while at the same time, adding a special post-processing step for cleaning all the elements that our pre-processing step is not able to manage due to the several non-structured (different format) pieces of text. By the end of this step, we have a nested lists object where (1) the first layer contains the 36 identified plays and (2) a second layer containing a 2-dimensional list containing both the name of the speaker and the spoken text associated with him/her. Therefore, we can easily identify the speaker and his/her parliament due to the simple perfect matching structure of the generated object.

Following the same logic as before, we can easily construct some regular expressions and apply them in a vectorized approach to our new (clean) object in order to extract the relevant information for generating a summary for each play: number of unique speakers, number of spoken chunks, number of sentences and words (we define a sentence as any piece of text finishing with a dot .; a question mark ? or an exclamation sign !; see details below).

Finally, summary plots and DataFrames are constructed in order to check possible trends in Shakespeare's plays and show our results.

Thus, we have the following steps:

i) Pre-processing (two steps)

ii) Plays separation

iii) Metadata extraction

iv) Cleaning and spliting

v) Summary: relevant data and plots

a) Plays: Extraction into a character vector/list

In order two perform our two-steps pre-processing operation, we define four auxiliary functions that will help us to keep an "easy-to-follow" code structure while keeping a high modularity and efficiency. The main objective of these functions is to perform the pre-process of the original .txt file based on a series of regular expressions that cover all the identified patterns inside the text file and generalize some non-vectorized functions into vectorized ones that can be applied to complex objects like nested lists, helping us to keep a non-verbose code and avoiding a higher error rate (due to code repetition).

The implementation is as follows:

a.1) toExtract() function: Scene directions pre-processing

1. Our first auxiliary function *toExtract.R* will process the majority of the scene directions inside the original text file. Exploiting the indentation and new lines structure, we are able to identify all the relevant scene direction chunks that are separated by (at least) one empty line from above are below of other elements. Important is to note that we are using '\r\n' as our new lines identifiers and the fact that we are explicitly declaring that our regular expressions should be used in a *multiline* mode. In other words, this special declaration will allow us to use the declared regular expression without limiting it to individual lines but to all the text.

   The input arguments of the functions consists of the original file in raw format (as is) after performing a classic *readLines()* operation on the .txt file and an important numerical input *num*. This numerical input allows us to detect different scene directions chunks based on their length/lines numbers. Therefore, we can easily perform a loop for different *num* values (when calling our function in the main script) in order to extract all stage direction chunks, adding flexibility to the operations (it can easily modified for other text structures).

   Hence, we are able to extract all the chunks inside the original file that follows the specific format identified, based on the structure of the file.

```
# Declaring the function
toExtract <- function(rawShakespeare, num){
  # Loading libraries
  library("stringr")

  # Pre-Process Extraction and cleaning
  DirPattern <- paste('^[\r\n]\\s{6,}([^\r\n&\\d<#\\[]*[\r\n]){1,',
                      as.character(num), '}\\s[\r\n]', sep = '')
  ToExtract <- str_extract_all(preprocShakespeareColl,
                               regex(pattern = DirPattern, multiline = TRUE))
```

2. However, as we already indicated in the general summary of our strategy, some elements/components inside the plays do not follow any specific (or repetitive) pattern, risking our extraction procedure. If we do not take this situation into account, we could end up extracting/deleting elements that we want

to keep. This is one of the trickiest and smartest pre-processing steps of our work since the "Songs" elements inside the plays do not have a different pattern when comparing them to some stage/scene directions.

In order to take this situation into account, we exploit the use of keywords. Based on our knowledge of the file structure and chunks components, we can easily identify a set of keywords/expressions that are part of regular scene directions in a repetitive way (an easy/explicit pattern). Thus, we will be able to select those already extracted lines (previous step) that have a match with any of the pre-defined keywords. Thanks to this approach, we can easily avoid committing mistakes such as deleting "Songs" from the pre-processed text file.

```
  # Pattern for filtering all the relevant scene directions based on a series
  # of keywords.
  KeywordsPattern <- paste('[\\b\\.\\s]*(([Ee]nters?)|(Exit)|(Here,?)|(Epitaph)|',
                           '(pass(es)?\\s)|([aA]larum)|([Tt]rumpets?)|(Sound)|',
                           '(sings\\s[^b])|(Falstaff)|(The\\stwo)|(throws)|([Ee]xeunt',
                           '[^\\.])|(Re\\-[eE]nter)|(fights)|([Ff]lourish)|',
                           '(come\\sforward)|([Gg]host)|(opens)|(Retreat)|',
                           '(After)|(cry)|(speaks)|(approaches)|(goes)|',
                           '(Apparition)|(comments)|(FAIRIES)|(appears)|',
                           '(ORATION)|(Grace)|(stand))[\\b\\.\\s]*', sep = '')
```

3. Therefore, we detect all the lines that match any of the previously defined keywords/expressions using the well-known *grep()* function, allowing us to filter the scene directions from other elements. In addition, since some scene directions are identical we add a second filtering step by just selecting those chunks which are unique in order to improve the performance of our pre-processing procedure.

```
  # Extraction of relevant scene directions without taking out SONGS
  FilteredToExtract <- ToExtract[[1]][grep(KeywordsPattern, ToExtract[[1]])]
  FilteredToExtract <- unique(FilteredToExtract)

  # Check the number of elements to be dropped (visualization purposes)
  length(FilteredToExtract)
```

4. Based on the filtered patterns for extraction identified, we can easily add the special characters '[' and ']' at the beginning and end of each chunk. Note that we can easily delete the identified chunks but instead, we are adding these extra characters as special starting/ending marks in order to keep all the original information of the plays while helping us to get rid of these elements in a future processing step.

   Finally, we return a processed text as an unique line string.

```
  # Scene directions are deleted from the text
  for (n in 1:length(FilteredToExtract)) {
    preprocShakespeareColl <- gsub(x = preprocShakespeareColl,
                                   pattern = FilteredToExtract[n],
                                   replacement = paste("[",
                                                       FilteredToExtract[n],
                                                       "]",
                                                       sep = ''),
                                   perl = TRUE)
  }

  # Return the processed object
  return(preprocShakespeareColl)
}
```

Therefore, the function returns a character object with pre-processed stage directions. From this, we will be able to separate the plays while keeping our extra marking characters inside the body of the text, for future processing/extraction.

a.2) PatternSubstitution() function: gsub wrap-up

1. The second function *PatternSubstitution()* is a simple wrap-up of the well-known *gsub()* function, including the *perl* syntax option with *TRUE* value by default (instead of false). Thanks to this simple implementation, we can easily construct all our regular expressions using the common Perl non-verbose syntax like \s for empty spaces, \w for words, \b for word separations (excluding punctuation marks), etc.

```
# Declaring the function
PatternSubstitution <- function(string, pattern, replace){

  # Looks for the pattern, replaces it and returns the new string
  string <- gsub(pattern, replace, string, perl = TRUE)
  return(string)
}
```

a.3) PreProcessing() functions: Adding special characters for processing

1. The core functions of the pre-processing steps are *PreProcessing()* and *PreProcessing2()*. The main difference between both consists of the type of element of the play that is being processed/modified: while *PreProcessing()* takes care of acts, scenes, characters and other special elements such as epilogues and prologues, *PreProcessing2()* takes care of a series of scene directions that cannot be covered by the *toExtract()* function due to their particular structure.

   As a first step, we eliminate all the beginning of the file without plays and the last play (Lover's Complaint) by hard-coding the lines containing these elements. After this procedure, we remove all the extra text (copyright declaration) by detecting the special characters << and >> as the starting and ending elements of these chunks. In addition, we split the character object (containing all the text) by lines and we unlist it in order to have a simple vector containing all the lines of the already processed file.

```
# Declaring the function
PreProcessing <- function(rawShakespeare){

  ## Calling external functions
  # Pattern substitution function
  source("PatternSubstitution.R")

  ## Pre-processing
  # Remove header, footer and sonnets based on lines
  rawShakespeare <- rawShakespeare[-(1:2815)]
  rawShakespeare <- rawShakespeare[-(121163:length(rawShakespeare))]

  # Remove extra text
  rawShakespeare <- paste(rawShakespeare, collapse = '\r\n')
  rawShakespeare <- gsub('<<[^>]*>>', '', rawShakespeare)
  rawShakespeare <- str_split(rawShakespeare, '\r\n')[[1]]
```

2. The main logic of these functions lies on a series of regular expressions that will allow us to modify/extract/add elements inside the plays such that we will be able to perform easy processing operations in future steps. In this particular function, we start by identifying all the elements associated with the ACTs, Scenes, Epilogues and Prologues. Note that one of the Prologues follows a specific speaker pattern that we are replacing after studying the structure of the file.

   We are adding the special characters && (Acts), # (Scenes), and %% (Epilogues).

```
## General Patterns and replacements: Add/Delete characters for easier processing
  # ACTs
  ACTPattern <- '^([Aa][Cc][Tt][_]?\\s?[a-zA-Z0-9]{1,3})\\.?'
  ACTReplace <- "&&\\1\\&&"

  # Scenes
  SCPattern <- '(\\b[S][Cc][Ee][Nn][Ee]\\b\\s+\\w+)\\.?'
  SCReplace <- '#\\1#'

  # Epilogues and Prologues
  EPPattern <- '(.*EPILOGUE)'
  EPReplace <- '%%\\1'

  PRPattern0 <- '^\\s{2}PROLOGUE\\.'
  PRReplace0 <- '<>QUINCE~'

  PRPattern <- '(.*PROLOGUE)'
  PRReplace <- '[\\1]'
```

3. Then, character (speaker) names are identified by a series of regular expressions. Note that we can easily mix some of them in order to obtain a larger expression with several conditions. However, we are avoiding this approach since it is not so easy to follow and can lead to misunderstandings (regular expressions can be very long and complicated), therefore, we are explicitly treating all different patterns/cases by an individual regular expression.

   As an example, in the first expression we are dealing with speakers with uppercase names while in the second expression we allow the possibility of having lowercase letters. A special case is identified inside the file, where some speaker names were not following any identified pattern and thus, we pre-process them separately from the rest (CHLCPattern1, CHLCPattern3, and CHLCPattern4).

   Important is to note that we are adding a special string (<>) just right next to (at the beginning) a speaker name. This will help us to perform all the future processing required to split the document by speakers and spoken chunks.

```
  # Characters: <> at the beginning for each character
  CHPattern1 <- paste('^\\s{1,2}([A-Z\',\\-]{2,}\\.(\\s+(&\\s)?[A-Z\\-]{2,}){1,})',
                      '\\.(\\s{0,2}[a-zA-Z\'\\[])', sep = '')
  CHReplace1 <- '<>\\1~ \\4'

  CHPattern2 <- paste('^\\s{1,2}([A-Z\',\\-]{2,}(\\s+(&\\s)?[a-zA-Z\'&,\\-]+){0,3})',
                      '\\.(\\s*[a-zA-Z\'\\[])', sep = '')
  CHReplace2 <- '<>\\1~ \\4'

  CHPattern3 <- '^\\s{2}([GW][A-Z]+)(\\s{2}\\[)'
  CHReplace3 <- '<>\\1~\\2'

  # Characters: lowercase characters special case (Cap. Wife. is reformated)
```

```
CHLCPattern0 <- '^\\s{2}([F]l[a-z]+)(.*)'
CHLCReplace0 <- '\\[\\1\\2\\]'

CHLCPattern1 <- '^(\\s{2}Cap)\\.(\\sWife\\.)'
CHLCReplace1 <- '\\1\\2'

CHLCPattern2 <- paste('^\\s{2}([A-Z][a-z]+(\\s{1}[A-Z][a-z]+){0,})\\.',
                      '(\\s{0,3}[iA-Z\'\\[\\(])', sep = '')
CHLCReplace2 <- '<>\\1~ \\3'

CHLCPattern3 <- '^\\s{4}((King)|(Queen)|(Luc))\\.(\\s((\'T)|([A-Z])))'
CHLCReplace3 <- '<>\\1~ \\5'

CHLCPattern4 <- '^\\s{2}(Fool)\\.(\\s{6}H)'
CHLCReplace4 <- '<>\\1~ \\2'
```

4. Special elements such as speaker names starting with numbers (e.g. “1. Car.”, “ 1. Play.”), different indentation (no initial indentation like for the Comedy of Errors play or with extra space at the beginning), and extra spaces after the marking dot (before the actual spoken text appears) are processed using powerful regular expressions. Important is to note that we are including the Comedy of Errors in our solution, raising the difficulty of the entire problem.

```
# Characters: Starting with number and lowercase cases
CHNPattern <- '^\\s{2}(\\d+\\.\\s[A-Za-z]+)\\.(.*)'
CHNReplace <- '<>\\1~\\2'

# Characters: Commedy of Errors special formats
CHCEPattern1 <- paste('^([A-Z\',\\-]{2,}(\\s+(&\\s)?[A-Z\'\\-]+){0,2})\\.(\\s{1,2}',
                      '[oA-Z\'\\[]\\s?[\',OHT!FSVLWa-z\\-])', sep = '')
CHCEReplace1 <- '<>\\1~\\4'

CHCEPattern2 <- paste('^\\s{4}([A-Z\',\\-]{2,}(\\s+(&\\s)?[A-Z\'\\-]+){0,2})\\.',
                      '(\\s{4}[oA-Z\'\\[])', sep = '')
CHCEReplace2 <- '<>\\1~\\4'

# Extra processing: Speakers with 3 to 5 spaces at the beginning
EXPattern1 <- '^\\s{3,5}([A-Z]{2,}(\\s[A-Z]+){0,})\\.(\\s[A-Z\'])'
EXReplace1 <- '<>\\1~ \\3'

# Extra processing: Special BOTH and MALVOLIO speakers with extra spaces after dot
EXPattern2 <- '^\\s{2}([A-Z]{2,})\\.(\\s+[A-Z\'\\[])'
EXReplace2 <- '<>\\1~ \\2'
```

5. Finally, we declare a series of regular expressions for (1) filtering the difficult Song chunks, (2) treat a series of difficult and non-structured expressions such as the AUTOLYCUS song, PROSPERO; TROILUS and CRESSIDA epilogues, (3) specific stage directions such as situations where speakers are reading a letter/document, singing a song, rising, etc., and (4) titles of the plays by adding the −− string at the beginning and end of the “by AUTHOR” sentence, fact that we are going to exploit in order to extract the name of each play.

```
# Songs
SONGPattern1 <- '([a-zA-Z\']+\\s)?S[Oo][Nn][Gg]\\.?)$'
```

```
SONGReplace1 <- '\\[\\1\\]'

SONGPattern2 <- '(\\s+(WINTER|SPRING))$'
SONGReplace2 <- '\\[\\2\\]'

# Special Song: Autolycus singing
SSONGPattern <- '\\[Enter\\sAUTOLYCUS,\\ssinging\\]'
SSONGReplace <- '<>AUTOLYCUS~'

# Special Epilogue: Prospero
SEPPattern <- '\\s+Spoken\\sby\\s(PROSPERO)'
SEPReplace <- '<>\\1~'

# Special Prologue
SPRPattern <- '\\s{8,}(TROILUS\\sAND\\sCRESSIDA)'
SPRReplace <- '<>\\1~'

# Titles
TIPattern <- '^(by.*)'
TIReplace <- '--\\1--'

# Special instructions (scenes/stage directions)
SPPattern1 <- '\\(((within)|([Ss]ings)|([Rr]eads)|(rises))([^\\)]*)\\)'
SPReplace1 <- '\\[\\1\\]'
```

6. Thus, we create a simple list containing all the patterns and their replacement expressions in order to process all of them in an automatic approach (loop).

```
# Patterns and replacements list
PatRepList <- list(
  c(ACTPattern, ACTReplace), c(SCPattern, SCReplace),
  c(EPPattern, EPReplace), c(PRPattern0, PRReplace0),
  c(CHPattern1, CHReplace1), c(CHPattern2, CHReplace2),
  c(CHPattern3, CHReplace3), c(PRPattern, PRReplace),
  c(CHLCPattern0, CHLCReplace0), c(CHLCPattern1, CHLCReplace1),
  c(CHLCPattern2, CHLCReplace2), c(CHLCPattern3, CHLCReplace3),
  c(CHLCPattern4, CHLCReplace4), c(CHNPattern, CHNReplace),
  c(CHCEPattern1, CHCEReplace1), c(CHCEPattern2, CHCEReplace2),
  c(EXPattern1, EXReplace1), c(EXPattern2, EXReplace2),
  c(SONGPattern1, SONGReplace1), c(SONGPattern2, SONGReplace2),
  c(SSONGPattern, SSONGReplace), c(TIPattern, TIReplace),
  c(SEPPattern, SEPReplace), c(SPPattern1, SPReplace1),
  c(SPRPattern, SPRReplace)
)
```

7. Hence, a substitution loop using our *PatternSubstitution()* function is performed for each pattern and its replacement expression in order to obtain a new processed text including all the mentioned modifications.

```
# Calling Substitution function for all patterns
for (n in 1:length(PatRepList)){
  Patt <- PatRepList[[n]][1]
```

```
    Repl <- PatRepList[[n]][2]
    rawShakespeare <- PatternSubstitution(rawShakespeare, Patt, Repl)
  }

  # Return pre-preprocessed Shakespeare
  return(rawShakespeare)
}
```

Therefore, we obtain a new text object with a series of extra characters inside that will help us for further processing. In addition, some special patterns were replaced in order to normalize their format.

8. The *PreProcessing2()* function follows the same structure as the previous one. In this case, no hard code modifications are performed since we have already deleted the beginning of the file and the last play (Lover's Complaint) as indicated above. Hence, we split the input raw text file by new lines separation.

```
# Declaring the function
PreProcessing2 <- function(rawShakespeare){

  ## Calling external functions
  # Pattern substitution function
  source("PatternSubstitution.R")

  # Split by new line
  rawShakespeare <- str_split(rawShakespeare, '\r\n')[[1]]
```

9. In this function, we deal with all the possible stage directions/information possible patterns inside the text file that were not captured by our *toExtract()* function. Therefore, we identify a series of keywords associated with regular scene directions such as: *Enter, Re-enter, [Ff]lourish, Exit, Exeunt,* etc., keywords that will allow us to include square brackets [and] at the beginning and ending of each scene direction/information, respectively. We include the Comedy of errors special formats in our solution.

```
  ## General Patterns and replacements: Add/Delete characters for easier processing
  # Stage Info: Scene directions adding '[' at the beginning and ']' at the end
  STGPattern1 <- paste('\\s*([<]?[^\\[](Exit)|(Enter)|(Re-enter)|(Exeunt)|(Flourish)|',
                       '(Here\\sa\\s)|(A\\slong\\sflourish,)\\.?)(.*)', sep = '')
  STGReplace1 <- '\\[\\1\\9\\]'

  STGPattern2 <- '^(\\s{20,}[a-z]+)(.*)'
  STGReplace2 <- '\\[\\1\\2\\]'

  STGPattern3 <- '^\\s{6,}([A-Z]{2,},)(.*)'
  STGReplace3 <- '\\[\\1\\2\\]'

  # Stage Info: Comedy of errors special format
  STCEPattern1 <- '^(((Exit)|(Enter)|(Re-enter)|(Exeunt)|(Flourish))\\.?)(.*)'
  STCEReplace1 <- '\\[\\1\\8\\]'

  STCEPattern2 <- '^(Exit|Enter|Re-enter|Exeunt)(.*)'
  STCEReplace2 <- '\\[\\1\\2\\]'
```

10. Next, we include all the special patterns identifying after an extensive exploratory study of the original text file, using the *grep()* function as suggested by the instructor in the problem statement. Here we deal

with scene directions with different indentation, other keywords such as *within, [Ss]ings, [Rr]eads, rises,* etc. Thanks to this exhaustive treatment, we are able to identify almost all possible scene directions (see appendix for more details regarding specific patterns that cannot be covered in a general way) fact that we are going to exploit in order to clean and extract information from the already processed object/file.

```
# Special instructions (scenes/stage directions)
  SPPattern1 <- '([A-Za-z\';:,!]\\s?\\.?)\\s{5,}(.*)\\.$'
  SPReplace1 <- '\\1         [\\2\\]'

  SPPattern2 <- '([A-Za-z\';:,!]\\s?\\.?)\\s{5,}([^\\[]+\\.)(\\[.*\\])$'
  SPReplace2 <- '\\1 [\\2\\] \\3'

  SPPattern3 <- '^\\s{18,}([A-Z]{2,}[^\\.\':]*)(?<!\\.,\';:)$'
  SPReplace3 <- '\\[\\1\\]'

  SPPattern4 <- '^\\s{30,}([^:#\\[]*(?<!([\',!;:])|([A-Z]\\.))$)'
  SPReplace4 <- '\\[\\1\\]'

  SPPattern5 <- '(THE\\sORDER.*)'
  SPReplace5 <- '\\[\\1\\]'

  SPPattern6 <- '^([a-z].*)'
  SPReplace6 <- '\\[\\1\\]'

  SPPattern7 <- '\\(((within)|([Ss]ings)|([Rr]eads)|(rises))([^\\)]*)\\)'
  SPReplace7 <- '\\[\\1\\]'

  SPPattern8 <- '^\\s{13}([A-Z]{2,})(.*)'
  SPReplace8 <- '\\[\\1\\2\\]'

  SPPattern9 <- '^\\s{8,}([Dd]ance\\.)'
  SPReplace9 <- '\\[\\1\\]'

  SPPattern10 <- '^\\s*([Tt]he\\s+[Tt]rumpet\\s+sounds?\\.)'
  SPReplace10 <- '\\[\\1\\]'

  SPPattern11 <- '([A-Z]+)\\)'
  SPReplace11 <- '\\1\\]'

  SPPattern12 <- '(They\\sfight\\.)(.*)'
  SPReplace12 <- '\\[\\1\\2\\]'

  SPPattern13 <- '(Here\\sthey\\se)(.*)'
  SPReplace13 <- '\\[\\1\\2\\]'
```

As before, we define the list of identified patterns and their replacement expressions in order to perform a substitution loop using our *PatternSubstitution()* function. Thus, as in the case of the *PreProcessing()* function, this function returns a processed text object, separated by lines.

a.4) Main Script A: Extract the plays into a character list

1. Using the previously defined functions, we develop a main script for pre-processing the original .txt file and return a character list object where each entry consists of a full Play (one of the 36 identified).

After indicating the current workdirectory, we load the required libraries for our code (*stringr* in our case) and then we proceed to load all the previously defined functions for using them inside the main script.

After loading them, we proceed to check if the original text file (pg100.txt) is available in the current working directory, if not, the file is downloaded from the given url and saved in the current working directory. Then, the raw text is read using the classic *readLines()* function, obtaining an object where each entry corresponds to one line of the file.

```
# Setting the working directory
setwd("~/ps3/")

# Loading libraries
library("stringr")

# Loading functions
source("PreProcessingVF.R")
source("toExtract.R")
source("PatternSubstitution.R")

# Check if the Shakespeare file is in the current directory
# If it does not exist: Download the file from given url
filename = "pg100.txt"
if (!file.exists(filename)) {
  url = "http://www.gutenberg.org/cache/epub/100/pg100.txt"
  download.file(url, filename)
}

# Read the file
rawShakespeare <- readLines(filename)
```

2. The first pre-processing step is performed. We invoke the *PreProcessing()* function and save the output into a new object. We can save the new object as a txt file in order to check the performed operations (for visualization purposes). We collapse this new object using \r\n as the new line entries and we explicitly transform the pre-processed object into a character object. This is an important step in order to be able to use multiline regular expressions, as in our *toExtract()* function.

   The final step of the first pre-processing step consists of invoking the *toExtract()* function for a series of different *num* values, allowing us to identify and extract all the different sizes (in terms of new lines) scene directions chunks that follows the explicit "empty-lines" pattern, as described in the section (a.1). Its range (2-15) is based on our knowledge of the original file's structure. Therefore, our current character object will have a series of extra characters for future processing: speakers, Acts, Scenes and the majority of Scenes directions have been processed at this point.

```
# Firts pre-process Step: Explicit scene directions
# Pre-Process the text using the PreProcessing function
preprocShakespeare <- PreProcessing(rawShakespeare)

# Debug: Print the current state of the file
write(preprocShakespeare, "preprocShakespeare.txt")

# Collapse the text into one long line
preprocShakespeareColl <- as.character(paste(preprocShakespeare, collapse = '\r\n'))

# Main cleaning loop: iterates over the length of the scene direction chunk
```

```
# without taking out the Songs inside the text
for (n in 2:15){
  preprocShakespeareColl <- toExtract(preprocShakespeareColl, n)
}
```

3. For the second pre-processing step, the *PreProcessing2()* function is invoked, obtaining a new character object that is collapsed again using \r\n for separating the lines. The current state of the object can be recorded for visualization (evolution check) purposes.

```
# Second pre-process step: More scene directions
# Pre-Process the text using the PreProcessing2 function
postprocShakespeare <- PreProcessing2(preprocShakespeareColl)
postprocShakespeare <- paste(postprocShakespeare, collapse = '\r\n')

# Save the current state of the file (Debugging purposes)
write(unlist(str_split(postprocShakespeare, '\r\n')) , "postprocShakespeare.txt")
```

4. Finally, all new lines characters are cleaned and we split the resulting object by plays using the special ending string inside the file "THE END". Hence, a 36 entries list is obtained: each entry corresponding to one play.

```
# Separate plays by endings: Delete newlines and split by endings obtaining
# a character vector
postprocShakespeareColl <- gsub('(\r\n)|(\r)|(\n)', ' ', postprocShakespeare)
PlaysShakespeare <- strsplit(postprocShakespeareColl,
                             '\\s+[-]?THE END[-]?\\s+')[[1]]

# Checking the total number of plays inside the file
length(PlaysShakespeare)

# Save information for visualization purposes
write(PlaysShakespeare, "ShakespearePlays.txt")
```

Therefore, we transform the original raw .txt file information into a list object including extra characters and modifications, ready for further processing operations, where each entry of the list contains a complete Shakespeare play.

b) Metadata: Extraction from each play

Thanks to our efficient and useful pre-processing steps, the extraction of metadata from each play can be easily performed using some simple auxiliary functions for both performing some list/vectorized operations and automatizing repetitive tasks, eliminating possible errors when copying the same code all the time. In this case, we developed two auxiliary functions that will help us to perform extraction-and-substitution operations inside our character list object via a very simple and convenient syntax and the possibility of recording/saving the elements into .txt files for visualization and checking purposes.

The implementation is as follows:

b.1) ExtractandSubs() function: Extraction and substitution

1. We define a simple function called *ExtractandSubs()*. This simple function allows us to invoke a substitution function like *gsub()* if all input values are provided or a simple extraction via the *str_extract_all()*

function. However, important is to note the fact that when the substitution version is invoked, a nested extraction is developed first: in other words, the substitution is being performed in those elements that match the extraction pattern. Therefore, we have a very simple but powerful function for performing both operations in a nested way, very useful for performing substitution operations in certain lines or elements from an object.

The input arguments consist of the "string" to be treated, the pattern that we want to extract "extracPatt", the pattern that we may want to replace "toreplacePat" and the substitution pattern that it is needed to be inserted in the matched lines/entries "subsPatt".

```
# Declaring the function
ExtractandSubs <- function(string, extractPatt, toreplacePat = NULL, subsPatt = NULL){

  # Loading libraries
  library("stringr")

  # Main operation: if subs patterns, call gsub. Otherwise, only str_extract
  if (!is.null(toreplacePat) & !is.null(subsPatt)){
    string <- gsub(toreplacePat, subsPatt, str_extract_all(string, extractPatt))
  }
  else {
    string <- str_extract_all(string, extractPatt)
  }

  # Return new string
  return(string)
}
```

b.2) PrintUtils() functions: Save your outputs, simple way

1. The *PrintUtils.R* file contains two functions, *List2Txt()* and *DF2Txt()*, that allow us to easily print out the results of a list (List2Txt) or a DataFrame (DF2Txt) inside a txt file. Thanks to these implementation, the user can easily check the metadata extracted and it can be stored for future references in order to not perform extra processing/calculations if the files (and thus, the information) are already available the current working directory.

   The *List2Txt()* uses a nested List object as the main input argument, alongside with a list containing all the filenames associated with each entry of the nested List object. By default, the separator pattern will be a new line. Then, the function perform a loop inside the nested Lists object in order to get access to all the individual elements of each embedded list, creating a .txt file for each one of them. In other words, the function can be understood as printing out separated columns of an abstract DataFrame into individual files, without creating a DataFrame. Note that no object is being returned (void function).

```
# Declaring the function
List2Txt <- function(Lists, filenames, OPTcollapse = '\n'){

  # Write the list as a collapsed column separating elements by a new line (default)
  for (n in 1:length(Lists)){
    write(paste(unlist(Lists[n]), collapse = OPTcollapse), filenames[n])
  }

}
```

2. On the other hand, the *DF2Txt()* function will transform a set of columns (nested list object, as with *List2Txt()* function) with their corresponding column names into a DataFrame. Then, the resulting

DataFrame will be printed out as a csv file (separated by comma by default) with the given name, as another input argument.

The operations are very similar to the ones performed in the previous function, noting that in this case we have to create the DataFrame object for convenience.

```
# Declaring the function
DF2Txt <- function(ColumnsArray, ColumnNames, filename, OPTSep = ','){

  # Loading libraries
  library("stringr")
  library("reshape2")

  # Split the columns by entries
  for (n in 1:length(ColumnsArray)){
    ColumnsArray[n] <- str_split(paste(unlist(ColumnsArray[n]),collapse = '---'),
                                 pattern = "---")
  }

  # Create the DF
  suppressMessages(DF <- melt(data.frame(ColumnsArray)))
  colnames(DF) <- ColumnNames

  # Save DF as a csv file (default)
  write.table(DF, filename, sep = OPTSep)
}
```

b.2) Main Script B: MetaData extraction

1. Using the previously defined functions, we develop a main script for extracting the metadata of each play inside our current pre-processed file. We set the current working directory, load the needed libraries and functions to be used inside the script. Then, we read the resulting pre-processed file after using the main script A as defined in the previous section and we check the length (integrity purposes) of the current object (36 lines). Note that if our character list object is available in the current session we do not need to read the .txt file since it is already loaded in memory, however, we are showing the "read-the-file" method for completeness and correctness of the code.

```
# Setting the working directory
setwd("~/ps3/")

# Loading libraries
library("stringr")

# Loading functions
source("ExtractandSubst.R")
source("PrintUtils.R")

# Read the Plays file and check its length
PlaysShakespeare <- readLines("ShakespearePlays.txt")
length(PlaysShakespeare)
```

2. Once the character list object is loaded, we proceed to the extraction of the relevant information inside each play. Using our pre-processing extra characters, we can easily determine the regular expressions

that are useful for detecting and extracting the values of interest. Therefore, we use our *ExtractandSubs()* function combined with the *lapply()* function in order to perform vectorized operations. Important is to note that two different operations are performed depending on if we need to obtain the actual strings (like years and titles) or the number (counting) of acts, scenes.

By the end of this step, we obtain four lists containing the metadata of each play inside the PlayShakespeare object.

```
## Extraction Process
# Years: Extraction pattern definition, extraction and recorded as a vector
YearPattern <- '^\\d{4}\\s+'
Years <- unlist(lapply(PlaysShakespeare,
                       function(x) ExtractandSubs(x, YearPattern, ' ', ''))
)

# Plays' Titles extraction
TitlePattern <- '([A-Z\',;]+\\s+){1,7}\\s+--by'
Titles <- unlist(lapply(PlaysShakespeare,
                        function(x) ExtractandSubs(x, TitlePattern, '\\s--by', ''))
)

# Number of Acts extraction
ACTSPattern <- '&&[^&]*&&'
NACTS <- unlist(lapply(PlaysShakespeare,
                       function(x) length(unique(unlist(ExtractandSubs(x, ACTSPattern)))))
)

# Number of Scenes extraction
SPattern <- '#[^#]*#'
NScenes <- unlist(lapply(PlaysShakespeare,
                         function(x) length(unlist(ExtractandSubs(x, SPattern))))
)
```

3. Following the same logic, we extract the body of each play, defined as all the content starting from the first Act and Scene.

```
# Play Body extraction
BodyPattern <- '\\&&[Aa][Cc][Tt]\\s+\\w+&&\\s+#[S][Cc][Ee][Nn][Ee]\\s[1Ii]#.*'
BodyPlay <- unlist(lapply(PlaysShakespeare,
                          function(x) ExtractandSubs(x, BodyPattern))
)
```

4. Finally, we invoke our *PrintUtils* functions in order to print out the extracted metadata into both separated text files per column and a summary reflected into one DataFrame containing all the relevant information per Play.

```
## Outputs: txt files for each relevant field
Lists <- list(Years, Titles, NACTS, NScenes, BodyPlay)
FileNames <- c("P1BYears.txt", "P1BTitles.txt", "P1BNACTS.txt",
               "P1BNScenes.txt","P1BBody.txt")
List2Txt(Lists, FileNames)

# Create DataFrame and save it as a txt/csv file
```

```
Cols <- list(Years, Titles, NACTS, NScenes)
ColNames <- c("Years", "Titles", "NActs", "NScenes")
DF2Txt(Cols, ColNames, "P1BDataFrame.txt")
```

Hence, by the end of the main script B, we obtain a series of lists containing all the relevant metadata of each play and a series of text files inside the current working directory for getting access to all this information and/or load it in next sessions.

c) Speakers and Text: Extraction from each play

The most challenging problem from the current set can be "easily" tackled thanks to our exhaustive pre-processing steps. In this case, we developed seven short and simple new auxiliary functions that will help us to keep a clean and easy-to-follow code while maintaining a good modularity of our general code implementation. A final cleaning (pos-processing) step is performed in order to eliminate elements of the plays that were not completely deleted in the previous steps due to the lack of a common pattern. All the functions are inside the *ListSubs.R* file.

Again, a series of text files are generated in order to be able to have access to them in future sessions without performing all the operations/calculations again every time the information is needed.

The implementation is as follows:

c.1) ExtractUnlist() function: Extraction in lists

1. In order to deal with lists operations such as extracting data, the *ExtractUnlist[]* function defines the correct syntax for accessing specific elements from a nested list using a set of indexes from another list. Therefore, it saves us time and allows us to avoid verbose syntax when performing this common operations (since we are working with nested lists in our solution).

   The main inputs are the original list (*Data*), the main *index* (first layer of the list) and the list of indexes to be extracted (*toextData*). The output consists of a list with all the extracted data.

```
# Declaring the function
ExtractUnlist <- function(Data, index, toextData){

  # Extract elements from a nested list based on indexes inside another list
  ExtractedData <- unlist(Data[index])[unlist(toextData[index])]

  # Return extracted Data
  return(ExtractedData)
}
```

c.2) DropUnlist() functions: Drop elements in lists

1. Similar to the previous function, in this case instead of extracting information from a nested list based on a list with indexes, the *DropUnlist()* function allows us to drop those particular indexes, while taking care of the unlist operations. Important is to note that if we do not have a matching element, the function will simply return the original list.

   The main inputs are the original list (*Data*), the main *index* (first layer of the list) and the list of elements to be dropped (*toextData*). The output consists of the original list without the dropped elements.

```r
# Declaring the function
DropUnlist <- function(Data, index, todropData){

  # Drop elements from a nested list based on indexes inside another list
  # if there are matching elements
  if (length(unlist(todropData[index])) > 0){
    NewData <- unlist(Data[index])[-unlist(todropData[index])]
  }

  # Otherwise, return the original Data
  else{
    NewData <- unlist(Data[index])
  }

  # Return new list without dropped data
  return(NewData)
}
```

The previous functions will be used as auxiliary functions in the next implementation, in order to simplify the code while keeping its modularity.

c.3) ListSubs() functions: Simple list operations

1. The *ListSubs()* function allows us to perform a series of vectorized substitutions using the already mentioned *Perl* syntax exploiting the *lapply()* function for keeping the list structure of our current working object. Thus, the function requires an original List (*ORList*), an original pattern that we want to replace (*ORPattern*), and a replacement pattern (*RPattern*). Note that using the *lapply()* function is very important in order to keep the original list structure of the object, returning a *NewList*

```r
# Declaring the function
ListSubs <- function(ORList, ORPattern, RPattern){

  # Loading libraries
  library("stringr")

  # Loading functions
  source("PatternSubstitution.R")

  # Apply the Substitution function to a list
  NewList <- lapply(ORList, function(x) PatternSubstitution(x, ORPattern, RPattern))

  # Return the new list
  return(NewList)
}
```

2. A similar implementation is done using the well-known *gsub()* function (without the *Perl* syntax flag) for simple comparison with the previous implementation.

```r
# Declaring the function
Listgsub <- function(ORList, ORPattern, RPattern){

  # Loading libraries
```

```
  library("stringr")

  # Apply the Substitution function to a list
  NewList <- lapply(ORList, function(x) gsub(ORPattern, RPattern, x))

  # Return the new list
  return(NewList)
}
```

3. A vectorized extraction function is declared inside *ListExt()*. This function allows the user to extract some information inside a series of lists ($ORList$, a nested list) based on a list of indexes ($ToExtIndexes$) containing the values of the entries we want to extract from each Play. It returns a list object containing the extracted elements from each list (play in our context). Note that we are invoking the *ExtractUnlist()* function in order to keep a short and easy to follow code, while avoiding indentation problems due to the amount unlisting operators needed for keeping the original format of our object.

```
# Declaring the function
ListExt <- function(ORList, ToExtIndexes){

  # Loading libraries
  library("stringr")

  # Loading functions
  source("ExtractUnlist.R")

  # Apply the Extract function to a list
  ExtList <- lapply(seq_along(ORList), function(x) ExtractUnlist(ORList, x, ToExtIndexes))

  # Return the extracted data (as a list)
  return(ExtList)
}
```

4. The *ListLines()* function uses the *grep()* function as a vectorized function in a nested lists object. The user provides a nested lists object ($ORList$) and a matching pattern for looking among all the elements inside of it. Then, it returns a simple list with all the line/entry values associated with the matched pattern. Pretty useful for extracting information from inside a nested list matching a certain pattern of interest.

```
# Declaring the function
ListLines <- function(ORList, Pattern){

  # Loading libraries
  library("stringr")

   # Apply the Extract function to a list
  LinesList <- lapply(ORList, function(x) grep(Pattern, unlist(x)))

  # Return the extracted data (as a list)
  return(LinesList)
}
```

5. Finally, the *ListDrop()* function allows the user to delete a series of indexes ($ToDropIndexes$) inside a nested lists object ($ORList$) by dropping them from the original element. We are using this function

in order to delete all the irrelevant information for the speakers/chunks extraction process such as Acts, Scenes, and Epilogues information. The function returns a new list object without the dropped elements.

Note that this function uses the previously defined *DropUnlist()* function as an auxiliary logic component.

```
# Declaring the function
ListDrop <- function(ORList, ToDropIndexes){

  # Loading libraries
  library("stringr")

  # Loading functions
  source("DropUnlist.R")

  # Apply the Drop function to a list
  NewList <- lapply(seq_along(ORList), function(x) DropUnlist(ORList, x, ToDropIndexes))

  # Return the new list)
  return(NewList)
}
```

c.4) Main Script C: Speakers and Text

1. Using the previously defined functions, we develop a main script for extracting the relevant information of the speakers and spoken chunks associated with each of them. As before, we set the current working directory and load all the relevant libraries and functions for the current script. Again, we load the resulting file from the previous main script (B in this case) by the classic *readLines()* function and we check its length for checking purposes.

```
# Setting the working directory
setwd("~/ps3/")

# Loading libraries
library("stringr")

# Loading functions
source("PatternSubstitution.R")
source("DropUnlist.R")
source("ExtractUnlist.R")
source("ListSubs.R")

# Read the Body file and check its length
PlaysBody <- readLines("P1BBody.txt")
length(PlaysBody)
```

2. The most relevant step for cleaning all the body of each play consists of defining the right regular expressions. Taking into account the extra characters that we have already introduced since the pre-processing steps, we can define three main cleaning patterns: (1) Delete all the elements inside brackets, allowing multiple occurrences of them (up to two nested bracket elements), (2) Delete all the elements inside brackets without eliminating the speakers information and (3) delete remaining brackets due to indentation.

   Once we define these "cleaning expressions", the script performs a simple loop over them and the *ListSubs()* function is applied in a vectorized manner, obtaining a clean *PlaysBody* object.

```
# Full cleaning []: Patterns are declared and concatenated
CPattern <- '\\[[^#\\[\\]>]*(\\[[^#>\\]\\[]*)?\\]([^#>\\[\\]]*\\])?'
CPattern2 <- '\\[[^<>\\]]*'
CPattern3 <- '\\s+(\\[)|(\\])\\s*'
CPatterns <- c(CPattern, CPattern2, CPattern3)

# Cleaning loop over all "cleaning" patterns
for (n in 1:length(CPatterns)){
  PlaysBody <- ListSubs(PlaysBody, CPatterns[n], '')
}
```

3. Then, remaining elements such as acts, scenes and epilogues are transformed to the speakers format in order to prepare our object for the next extraction & dropping step, exploiting the fact that we are going to split the current object by the speakers separator <>. Again, a simple loop is performed covering all the declared patterns.

```
# Extract dialog chunks after taking out the ACT headers AND Epilogues
ACTPattern <- '&&A'
ACTReplace <- '<>A'
EPIPattern <- '%%'
EPIReplace <- '<>'
SCPattern <- '(\\s*)#(S[Cc][Ee][Nn][Ee])'
SCReplace <- '\\1<>\\2'

AESPatterns <- list(c(ACTPattern, ACTReplace), c(EPIPattern, EPIReplace),
                    c(SCPattern, SCReplace))

# Replacing loop over all patterns
for (n in 1:length(CPatterns)){
  PlaysBody <- ListSubs(PlaysBody, AESPatterns[[n]][1], AESPatterns[[n]][2])
}
```

4. After cleaning the plays, we proceed to split it by the characters special inserted expression <>, obtaining a new nested list. Note that the first layer contains the 36 plays, while the second layer contains all the spoken chunks separated by speakers, acts, scenes, and epilogues (remembering the previous step)

```
# Split the Body by spoken chunks
PlaysBody <- lapply(PlaysBody, function(x) str_split(x, '<>'))
```

5. Next, we drop all the acts, scenes and epilogues information using both *ListLines()* and *ListExt()* functions. The general approach consists of detecting those lines (across the entire nested list), extract its information (possible useful) and then drop those elements from our main text object, cleaning it from them. Here we can easily understand the power and utility of our auxiliary functions, allowing us to keep a very easy to follow and non-verbose code for developing some complex operations in order to keep the object original structure.

   Finally, we drop possible empty lines.

```
## Dropping Acts/Scenes, Epilogue and empty strings
# Detect lines with ACTs and Scenes for dropping them
LNACTPatern <- '^([aA][cC][tT])|(S[Cc][Ee][Nn][Ee])'
```

```
ACTDrop <- ListLines(PlaysBody, LNACTPatern)

# Extract Acts/Scenes lines (visualization purposes)
InfoACTScenesDrop <- ListExt(PlaysBody, ACTDrop)

# Drop Acts/Scenes lines from the Body
PlaysBody <- ListDrop(PlaysBody, ACTDrop)

# Detect lines with Epilogues for dropping them
EPDPattern <- '.*EPILOGUE'
EpilogueDrop <- ListLines(PlaysBody, EPDPattern)

# Extract Epilogue lines
InfoEpilogues <- ListExt(PlaysBody, EpilogueDrop)

# Drop Epilogue lines
PlaysBody <-  ListDrop(PlaysBody, EpilogueDrop)

# Empty Space
PlaysBody <- lapply(seq_along(PlaysBody),
                    function(x) {
                      unlist(PlaysBody[x])[unlist(PlaysBody[x]) != ""]
                    }
                    )
```

6. Due to some inconsistencies when adding stage directions and other elements to the plays and the lack of a common pattern along them, a final cleaning step is performed. Based on a series of keywords and our knowledge of the structure of each file created during the current solution approach, we are able to identify those extra elements inside the current object lines. Therefore, a series of regular expressions are defined in order to get rid of this extra information, using the fact that almost all the remaining directions are at the end of their lines, simplifying the construction of the useful regular expressions.

   All the patterns are concatenated as a large vector in order to perform a cleaning loop in the next step.

   As an extra comment, here we are dealing with non-common objects(chunks) like the one denominated as "THE VISION" and the directions for the "THE ORDER OF THE CORONATION". Hence, we end up with an almost (see appendix for details) completely clean Body for each play.

```
# Final cleaning step: Treatment by line for special scene directions
# Special Patterns are declared
SPPat1 <- '\\s+(Sound\\sa\\sflourish.*)'
SPPat2 <- '\\s+(He\\stakes\\sup.*)'
SPPat3 <- '\\s+(The\\sLORDS\\srise.*)'
SPPat4 <- '\\s+(During\\sthis\\ssong.*)'
SPPat5 <- '\\s+(\\d\\.\\sA\\slively\\sflourish.*)'
SPPat6 <- '\\s+(THE\\sVISION\\..*)'
SPPat7 <- '\\s+Then\\s+enter([^,]*,){3}'
SPPat8 <- '\\s+TITUS\\s+kills\\s+him.*'
SPPat9 <- '\\s+courteously\\sto\\severy.*'
SPPat10 <- '\\s+Ross,\\sand\\sW.*'
SPPat11 <- '\\s+before\\shim;[^A-Z]*S.*'
SPPat12 <- '\\s+The\\sdrum\\sp.*'
SPPat13 <- 'CARDINAL\\sB.*'
SPPat14 <- 'LA\\sPUCELLE\\sand\\sYORK\\sfight.*'
```

```
SPPat15 <- '\\s{10,15}SIR\\s+[A-Z].*'
SPPat16 <- 'The\\sCAPTAINS.*'
SPPat17 <- 'OF\\sTHE\\sP.*'
SPPat18 <- '\\sEPHESUS,.*'
SPPat19 <- '\\s+DROMIO\\sOF\\sSYRACUSE\\sas.*'
SPPat20 <- '\\s{2,}enter\\s[A-Z].*'
SPPat21 <- '\\s+Here,\\safter.*'
SPPat22 <- '\\s+The\\sbattle\\sc.*'
SPPat23 <- '\\s+Iago\\sfrom.*'
SPPat24 <- 'CYMBELINE,.*'

FCPatterns <- c(SPPat1, SPPat2, SPPat3, SPPat4, SPPat5, SPPat6, SPPat7, SPPat8,
                SPPat9, SPPat10, SPPat11, SPPat12, SPPat13, SPPat14, SPPat15,
                SPPat16, SPPat17, SPPat18, SPPat19, SPPat20, SPPat21, SPPat22,
                SPPat23, SPPat24)
```

7. For visualization purposes, the current cleaned object is copied and the final cleaning loop is performed in this new object, obtaining a new list without all the matched elements. Note that we are using the help of our *Listgsub()* function for implementing this operation.

```
# New List for cleaning: original format is kept as PlaysBody list
CleanPlays <- PlaysBody

# Cleaning loop
for(n in 1:length(FCPatterns)){
  CleanPlays <- Listgsub(CleanPlays, FCPatterns[n], '')
}
```

8. Finally, we separate each chunk (inside each play) by speaker and text, taking advantage of an extra character added at the beginning of the pre-processing step. Therefore, right now we have a nested list with the first layer containing all the plays (36) and then a second layer for each match between speaker & spoken chunk.

   The final file is recorded following two different formats that can be useful for further analysis, we include them as examples for visualization purposes. Some output examples are provided.

```
# Separate each spoken chunk and speaker: nested lists for a perfect match
# between speaker and spoken chunk
CleanPlaysSep <- lapply(seq_along(CleanPlays),
                        function(x) (str_split(unlist(CleanPlays[x]), "~")))

# Output Files
# Save the clean body file as a txt
write(unlist(CleanPlays), "CleanPlaysBody.txt")

# Save the speakers/chunk separated clean body as a txt
write(unlist(CleanPlaysSep), "SpeakersChunks.txt")

# Output Examples: [[NPlay]][from x to y lists containing speakers and text]
CleanPlaysSep[[1]][1:2]
CleanPlaysSep[[15]][2:4]
CleanPlaysSep[[36]][10:13]
```

Thus, at this point we have a clean character object (list of lists) containing a perfect match for each speaker and text, classifieds by Plays.

d) Summary Statistics: Information from each play

At this point, we no longer need to define extra auxiliary functions for processing the clean file. In this case, the approach is a very straightforward implementation of a series of *lapply* applications of different functions, therefore, we are not creating separating functions for each procedure in order to perform all the relevant operations inside the main script associated with this problem. Hence, computations are performed directly on the clean object obtained as the main output from the main script C: the nested list with a first layer for plays and then a series of perfect matching vectors associating speakers and text respectively.

A summary .txt file is created by the end of the script based on a DataFrame object generated from the individual calculations performed. Information is displayed to the screen for showing purposes.

The implementation is as follows:

d.1) Main Script D: Summary per play

1. As usual, we start by setting the current working directory and loading the needed libraries and functions.

```
# Setting the working directory
setwd("~/ps3/")

# Loading libraries
library("stringr")

# Loading functions
source("ExtractUnlist.R")
```

2. The first calculations are associated with the number of speakers. First, we create a nested list containing all the indexes corresponding to speakers. This is pretty simple if we notice that they will be located in the odd positions after unlisting the object, and hence, we can easily obtain one list of indexes per Play. Then, in order to obtain the Unique speakers we extract the lines associated with the *SpeakerIndexes* obtained before and we normalize them by transforming all letters into uppercase letters and then using the *unique()* function for filtering repetitions. Thus, we simply apply the *length()* function to the new object and we obtain the total number of unique speakers per play as a numeric list object.

   Important is to note that we can easily calculate the total number of spoken chunks by just counting the second layer elements per play or (as we show in this code) by dividing the total length of the unlisted play and divide it by two, since we have a perfect match between speaker and text.

```
## Summary per play
# Number of unique speakers: Extract unique speakers and count them
SpeakerIndexes <- lapply(CleanPlaysSep, function(x) seq(1, length(unlist(x)), 2))
UniqueSpeakers <- lapply(seq_along(CleanPlaysSep),
                         function(x)
                           unique(
                             str_to_upper(ExtractUnlist(CleanPlaysSep, x, SpeakerIndexes))
                           )
                         )
NUniqueSpeakers <- lapply(UniqueSpeakers, function(x) length(unlist(x)))

# Number of spoken chunks
NSpoken <- lapply(CleanPlaysSep, function(x) length(unlist(x)) / 2)
```

3. Following the same logic, we can easily compute the indexes associated with text (not speakers) by noticing that they will be in the even entries of the list, after unlisting it. Hence, we obtain a list of indexes *ChunkIndexes* associated with the text inside the plays.

   After this, we define a regular expression for filtering sentences, defined as every piece of text finishing with a dot, question mark or exclamation mark. Then, we simply define a function "on-the-fly" where we extract the lines containing text from each play and then we extract all the matching (sentences) within them. Finally, we simply unlist them in order to obtain a list with out original format, containing all sentences (as we defined before) per Play (two layers list). Again, we simply compute the total number of sentences by applying the *length()* function inside an *lapply()* environment.

```
# Number of sentences
ChunkIndexes <- lapply(CleanPlaysSep, function(x) seq(2, length(unlist(x)), 2))
SentencePattern <- '[^\\.\\?!]+[\\.\\?!]'
Sentences <- lapply(seq(CleanPlaysSep),
                        function(x) {
                          unlist(
                            str_extract_all(
                              ExtractUnlist(CleanPlaysSep, x, ChunkIndexes),
                              SentencePattern)
                          )
                        }
                    )
NSentences <- lapply(Sentences, function(x) length(unlist(x)))
```

4. Exactly as before, we define a word pattern such that a word can contain any number (at least one) of letters and/or apostrophes. Note that instead of taking out the punctuation marks we are using the \b expression, supported by the *Perl* syntax. Again, we compute the number of words by simply applying a *length()* function with *lapply()*.

```
# Number of words (all)
WordsPattern <- '\\b[A-Za-z\']+\\b'
Words <- lapply(seq(CleanPlaysSep),
                    function(x){
                      unlist(
                        str_extract_all(
                          ExtractUnlist(CleanPlaysSep, x, ChunkIndexes),
                          WordsPattern)
                      )
                    }
)
NWords <- lapply(Words, function(x) length(unlist(x)))
```

5. For calculating the average number of words per chunk, per play, we can take advantage of the number of words and number of spoken text lists previously computed. Therefore, we can easily compute the average using a simple function inside an *lapply()* function. As a detail, we set the precision of the numbers up to three decimals.

   Finally, we compute the total number of unique words by simply extract the unique words inside the word list (for each play) and then computing the corresponding length of the resulting list for each play.

```
# Average number of words per chunk: 3 decimals
AverageWordsChunk <- lapply(seq(NWords),
                            function(x) unlist(NWords[x]) / unlist(NSpoken[x]))
```

```
AverageWordsChunk <- gsub('(\\d+\\.\\d{3}).*', '\\1', AverageWordsChunk)

# Unique words (no repetitions)
UniqueWords <- lapply(Words, function(x) unique(str_to_lower(unlist(x))))
NUniqueWords <- lapply(UniqueWords, function(x) length(unlist(x)))
```

6. We output the results to the main console for visualization purposes.

```
## Display results (summary)
# Unique Speakers
UniqueSpeakers
unlist(NUniqueSpeakers)

# Number of chunks
unlist(NSpoken)

# Number of sentences
unlist(NSentences)

# Number of words (all)
unlist(NWords)

# Average number of words per chunk: 3 decimals
AverageWordsChunk

# Unique words (no repetitions)
unlist(NUniqueWords)
```

7. In order to keep all the results of the performed operations while helping the user to visualize all the information gathered, we print out all the summary information into a .txt file (full information). In the next section, we will also create a DataFrame containing all this information and we will record it for getting a less verbose output (simplest summary).

```
# Save all the information to a .txt file
ToSummary <- list(UniqueSpeakers, unlist(NUniqueSpeakers), unlist(NSpoken),
                  unlist(NSentences), unlist(NWords), AverageWordsChunk,
                  unlist(NUniqueWords))
Separators <- c('--------------------UniqueSpeakers--------------------',
                '--------------Number of unique Speakers--------------',
                '---------------Number of Spoken Chunks---------------',
                '----------------.-Number of sentences-.----------------',
                '-----------------Number of Words (Total)--------------',
                '----------Average number of words per chunk----------',
                '------------------Number of Unique Words--------------')

# Create the txt file (loop over separators and information)
for (n in 1:length(ToSummary)){
  write(Separators[n], "Summary.txt", append = TRUE)
  write(unlist(ToSummary[n]), "Summary.txt", append = TRUE)
}
```

Therefore, by the end of the main script D we will have all the summary statistics from each play in a series of list objects. In addition, a text file with all the relevant information is created.

e) Descriptive analysis: Plot and Trends

As a final analysis of our Shakespeare's plays processing, we would like to visualize the summary statistics as a function of time such that we could possibly identify any trends in Shakespeare's plays over the course of his writing career. In order to provide a clean and concise code, we create an auxiliary function for generating nice looking plots using a simple line of code for each plot needed. Moreover, a DataFrame with all the summary statistics is created and recorded as a .txt file in order to have future access to that information and for visualization purposes.

The implementation is as follows:

e.1) plotSummary() function: Easy and nice plotting

1. Our "easy-to-plot" function will plot any variable Y as a function of a variable X that in this case will be the years associated with Shakespeare's plays. The DataFrame *df* containing these columns is the third argument of the function. Then, the label of the x-axis is set to "Years" as default and a label for the y-axis is required (string). Finally, the user can pass a color flag for the plot, that will be red by default.

   As usual, we load the required libraries. In this case, we are using the classic *ggplot2* library for generating nice plots.

```
# Declaring the function
plotSummary <- function(X, Y, df, xlab = "Years", ylab, color = "red"){

  # Loading libraries
  library(ggplot2)
```

2. The plot is initialized and a series of theme modifications are performed in order to obtain a better looking (than default) plot. In this case, we modify the font types and sizes for the title, axis and potential legend. Upper right and top axis are deleted for visualization purposes and a light gray grid is set as a background for our plots.

```
  # Defining the plot
  plotfile <- ggplot() +

    # Modify the borders, grid and lines colors and format
    theme(axis.line = element_line(size = 1, colour = "black"),
          panel.grid.major = element_line(colour = "#d3d3d3"),
          panel.grid.minor = element_blank(),
          panel.border = element_blank(), panel.background = element_blank())+
```

3. The lines between the relevant points are declared and the labels of the x-axis are rotated in 90° degrees in order to obtain a legible plot. Then, the title of the plot is defined depending on the arguments provided by the user.

```
# Add the line to the graph
    geom_line(aes(y = Y, x = X, group = 1), size = 1.2,
              data = df, stat = "identity", colour = color, alpha = 0.5) +

    # Vertical labels for x-axis
    theme(axis.text.x = element_text(angle = 90, hjust = 1)) +
```

```
    # Check if we have explicit labels: define labels and title
    if ((!is.null(xlab)) & (!is.null(ylab))){
      labs(x = xlab, y = ylab,
           title = paste(ylab, " in Shakespeare plays vs ", xlab, sep = ''))
    }
    # Else, a generic name is given
    else{
      labs(title = paste("Number of Y in Shakespeare plays vs Years", sep = ''))
    }
```

4. Finally, the plot is generated.

```
  # Plot the current graph
  plotfile
}
```

Hence, thanks to this function we can easily generate a series of nice looking plots by just invoking the function in one simple line, keeping our main script very simple.

e.2) Main Script E: Plots and trends

1. As usual, we start by setting the current working directory and loading the needed libraries and functions.

```
# Setting the working directory
setwd("~/ps3/")

# Loading libraries
library("reshape2")
library("ggplot2")

# Loading functions
source("plotSummary.R")
```

2. We create a DataFrame with all the relevant statistics obtained so far. Each list will be a column inside our Summary DataFrame. The DataFrame is recorded as a .csv file, using commas as separators between column values.

```
## Creating Data Frame
# Columns are defined
Columns <- list(Years, Titles, NACTS, NScenes, NWords, NUniqueSpeakers,
                NSpoken, NUniqueWords, AverageWordsChunk)

# Format is applied to each columns
ProcCols <- lapply(Columns, function(x) str_split(paste(x, collapse = '---'), "---"))

# DataFrame is created and columns names are assigned
suppressMessages(SummaryDF <- melt(data.frame(ProcCols)))
colnames(SummaryDF) <- c("YEAR", "TITLE", "ACTS", "NSCENES", "NWORDS", "NSPEAKERS",
                         "NCHUNKS", "NUNIQUE_WORDS","AVG_WORDS")

# Save the DataFrame as txt file
write.table(SummaryDF, "P2EDataFrame.txt", sep = ",")
```

3. Finally, we simply call the *plotSummary()* function for each plot we want to obtain. Important is to note that since we have some years with more than one play, we are showing all points in our plot in order to explicitly indicate that our analysis has been performed over all 36 plays and no inconsistencies are involved. *Depending on the fonts installed in the computer, a series of warnings can be obtained after generating the plots. An extra line with $options(warn = -1)$ can be included.

```
# Plots using the plotSummary function
plotSummary(SummaryDF$YEAR, SummaryDF$NSCENES, SummaryDF,
            y = "Number of Scenes", color = "red")
```

```
plotSummary(SummaryDF$YEAR, SummaryDF$ACTS, SummaryDF,
            y = "Number of ACTs", color = "blue")
```

```
plotSummary(SummaryDF$YEAR, SummaryDF$NWORDS, SummaryDF,
            y = "Number of Words", color = "#40b8d0")
```

```
plotSummary(SummaryDF$YEAR, SummaryDF$NSPEAKERS, SummaryDF,
            y = "Number of Speakers", color = "#00b33c")
```

```
plotSummary(SummaryDF$YEAR, SummaryDF$NCHUNKS, SummaryDF,
            y = "Number of Chunks", color ="#e64d00")
```

```
plotSummary(SummaryDF$YEAR, SummaryDF$NUNIQUE_WORDS, SummaryDF,
            y = "Number of Unique Words", color ="#751aff")
```

```
plotSummary(SummaryDF$YEAR, SummaryDF$AVG_WORDS, SummaryDF,
            y = "AVG Number of words", color = "#ff6666")
```

Problem 3

Class: general description

In order to design our Object-Oriented Programming (OOP) approach to the Shakespeare analysis proposed in the previous section, we will develop a class definition in the context of a formal Python class. Therefore, our class will follow a classic Python structure and methods/function paradigm.\

The main idea consists of creating a parent Class *ShakespearePlays* that will be able to manage objects from the *Plays* Class, an specific class with the capacity of manipulating independent plays objects. Thus, we will define two main classes in order to cover all the relevant elements from the previous analysis (and more). \

For completeness, we will perform a detailed description of each of the relevant fields in order to understand the brief explanation associated with each of the described methods in the next section.

a.1) Class ShakespearePlays: Fields

In order to provide more flexibility to the users, we can allow them to initialize a ShakespearePlays class object by two different methods (see next section for full details):

1. By providing a valid .txt file such as the one used in the previous problem. The file must follow the same general structure as the already mentioned text file, including as many plays as the user wants to analyze. Thus, this approach follows exactly the structure of the previous problem, allowing us to use our already defined functions as class' methods.
2. As a second option, the user may provide an object list/vector containing all the *Plays* class objects that he/she wants to analyze. In this case, the processing will be easier and more flexible (if well implemented) since all methods from the *Plays* class will be accessible.

Therefore, our constructor method (*def ___init___* in Python context) should be able to detect the initial input and perform different operations depending on its type (trivial implementation with conditional clauses). All relevant and basic fields such as the number of plays, their titles or year of publication will be calculated inside the constructor method (see next section for details) while demanding operations (both in terms of memory and computational time) such as pre-processing and cleaning the body of the plays or detecting the unique number of words/speakers will not be calculated at the beginning (inside the constructor method) waiting for the explicit user instruction — by calling specific methods — in order to perform those demanding computations.

Fields

- ***RawPlays***: Character list object containing all the raw format from a text file (as in the previous problem) separated by lines. The number of entries consists of the total number of lines in the original raw file. (character list/vector)

  Required for constructor: Yes unless ***ObjectPlaysList*** is provided.

- ***PreProcPlaysList***: Character list containing all the plays inside the *RawPlays* character object as long string characters, right after the application of the pre-processing operations (see methods below). In this case, there is one entry per initialized play instead of one entry per line of the text file and extra characters have been already inserted inside the plays bodies for extraction purposes. This is an example of a field that is not going to be initialized inside the constructor. (character list/vector)

  Required for constructor: No.

Initial Value: Computed inside the constructor.

- ***ObjectPlaysList***: If the class is initialized using objects from the *Plays* Class instead of a raw .txt file, this field consists of an object list, where each entry is associated with a *Plays* Class object. (object list/vector)

  Required for constructor: Yes unless ***RawPlays*** is provided.

- ***CleanPlaysBody***: Character list containing all the processed (cleaned) initialized plays, ready for statistical information extraction. Each entry consists of a long string containing the body of each play. Equivalent to the CleanPlays list obtained in our code for the previous problem. (character vector/list)

  Required for constructor: No.

  Initial Value: Null.

- ***SpeakerTextPlays***: Character nested list (more than one layer) where the first dimension contains as many entries as the total number of initialized plays while the second consists of the total number of spoken chunks/speakers. The last layer contains both the name of the speaker and the corresponding spoken text. (character nested list)

  Required for constructor: No.

  Initial Value: Null.

- ***NPlays***: Numeric value containing the total amount of initialized plays objects. (numeric [int])

  Required for constructor: No.

  Initial Value: Computed inside the constructor.

- ***Titles***: Name/Title of the plays character list associated with each of the initialized plays when creating the ShakespearePlays object (character vector/list)

  Required for constructor: No.

  Initial Value: Computed inside the constructor.

- ***Years***: Year of publication numeric/character list associated with each of the initialized plays when creating the ShakespearePlays object (numeric [int] vector/list)

  Required for constructor: No.

  Initial Value: Computed inside the constructor.

- ***NActs***: Numeric list containing the number of acts of each initialized play. (numeric [int] vector/list)

  Required for constructor: No.

  Initial Value: Computed inside the constructor.

- ***NScenes***: Numeric list containing the number of scenes of each initialized play. (numeric [int] vector/list)

  Required for constructor: No.

  Initial Value: Computed inside the constructor.

- ***NSpeakers***: Numeric list containing the number of unique speakers of each initialized play. All elements in uppercase. (numeric [int] vector/list)

  Required for constructor: No.

  Initial Value: Null.

- ***NUnwords***: Numeric list containing the number of unique words contained in each initialized play. (numeric [int] vector/list)

  Required for constructor: No.

  Initial Value: Null.

- ***NWords***: Numeric list containing the number of words inside of each initialized play. (numeric [int] vector/list)

  Required for constructor: No.

  Initial Value: Null.

- ***Speakers***: Character list containing the name (string) of the unique speakers of each initialized play. (character list/vector)

  Required for constructor: No.

  Initial Value: Null.

- ***UniqueWords***: Character list containing the unique words (string) present inside of each initialized play. (character list/vector)

  Required for constructor: No.

  Initial Value: Null.

- ***Words***: Character list containing the all the words (string) present inside of each initialized play. (character list/vector)

  Required for constructor: No.

  Initial Value: Null.

- ***NChunks***: Numeric list containing the number of spoken text (chunks) for each initialized play. (numeric [int] list/vector)

  Required for constructor: No.

  Initial Value: Null.

- ***AVGWordsChunks***: Numeric list containing all the average number of words per spoken chunks for each initialized play. (numeric [float] list/vector)

  Required for constructor: No.

  Initial Value: Null.

- ***SummaryTable***: Dataframe containing the summary statistics of all the plays that have been initialized. (DataFrame containing numeric [int,float] and character columns)

  Required for constructor: No.

  Initial Value: Null.

- ***NcharsPlays***: [Extra information] Numeric list containing the total number of characters of each initialized play. (numeric [int] vector/list)

  Required for constructor: No.

  Initial Value: Computed inside the constructor.

- ***CleanedPlays***: [Extra information] Boolean list containing ones when a Play has been cleaned and zeros otherwise. This will allow us to keep track of the user operations such as cleaning only a subset of the total amount of plays available using the *clean_Plays()* method as defined below. (numeric [int] vector/list)

  Required for constructor: No.

  Initial Value: All entries equal to 0.

- ***ID***: [Extra information] Numeric list containing a specific number associated with a Play, for selection/extraction purposes. (numeric [int] vector/list)

  Required for constructor: No.

  Initial Value: Computed inside the constructor.

---

a.2) Class Plays: Fields

Clearly, this class will have all the previous fields in an individual setting: e.g. instead of *RawPlays* it will have a *RawPlay* field; *Titles* will be *Title*, and so on.\

On the other hand, fields such as *ObjectPlaysList* and *NPlays* are no longer useful and can be omitted.\

Thanks to this structure, we will allow the main class *ShakespearePlays* to easily handle objects from within this class, adding flexibility to our OOP framework.

Fields

b.1) Class ShakespearePlays: function, inputs, modifications and outputs.

In this section we present a series of methods that will be useful to perform all the operations needed for developing an analysis like the one presented in section 2. For all of them, we will assume that we have access to all the functions and scripts from the previous problem, and thus, all operations inside the method are valid and correct. \

Clearly, more methods can be added in order to add more flexibility to the main class. In addition, *Plays* class elements could been initialized from inside the *ShakespearePlays* Class. However, we focus our development on the core functionality. \

We can also assume the fact that all the following methods can handle objects created in the *Plays* class such that dealing with them is natural and efficient. Following this logic, all methods from the *Play* class

can be applied to these elements inside the following methods in order to easily get access to all the relevant information for the method in the case that we are dealing with these kind of objects.

Methods

- ***___init___()***: Constructor method, generates the new object from the class based on an input that can be a character list (for the *RawPlays* approach) or an object list (*ObjectPlaysList* approach) containing elements from the *Plays* Class. Initializes a series of fields and does not return any element. (constructor/void method)

  1. Inputs: (character list) or (object list)
  2. Fields created: *RawPlays* or *ObjectPlaysList*, *PreProcPlaysList*, *NPlays*, *Titles*, *Years*, *NActs*, *NScenes*, *NcharsPlays*, *CleanedPlay*, *ID*.
  3. Field modified: Null.
  4. Outputs: Null, (void method).

- ***get_NPlays()***: returns the total number of plays to the user. (returns numeric [int] value)

  1. Inputs: Null.
  2. Fields created: Null.
  3. Field modified: Null.
  4. Outputs: (numeric [int]: *NPlays*)

- ***get_Titles(****nplays*[] = *NPlays)*: returns a character list containing all the titles associated with the plays given by the user via a numeric list. By default (if no values are given by the user) it returns a character list with the title of each identified/initialized play inside the object. Clearly, it will prompt an error message if the given input does not follow the specified format and/or the user asks for a non-existing (out of the bounds) play. (returns character list)

  1. Inputs: (numeric [int] list: *ID*) or Null.
  2. Fields created: Null.
  3. Field modified: Null.
  4. Outputs: (character list: *Titles*)

- ***get_Years(****nplays*[] = *NPlays)*: returns a character list with the year of publication of the selected plays inside the object. (returns character list)

  1. Inputs: (numeric [int] list: *ID*) or Null.
  2. Fields created: Null.
  3. Field modified: Null.
  4. Outputs: (character list: *Years*)

- ***get_NActs(****nplays*[] = *NPlays)*: returns the total number [int] of initialized plays inside the current object. (returns numeric [int] list)

1. Inputs: (numeric [int] list: *ID*) or Null.
2. Fields created: Null.
3. Field modified: Null.
4. Outputs: (numeric [int] list: *NActs*)

- ***get_NScenes(****nplays*[] = *NPlays)*: returns a numeric [int] list containing the number of scenes of the selected plays inside the object. (returns numeric [int] list)

  1. Inputs: (numeric [int] list: *ID*) or Null.
  2. Fields created: Null.
  3. Field modified: Null.
  4. Outputs: (numeric [int] list: *NScenes*)

- ***get_NcharsPlays(****nplays*[] = *NPlays)*: returns a numeric [int] list containing the number of characters of the selected plays inside the object. (returns numeric [int] list)

  1. Inputs: (numeric [int] list: *ID*) or Null.
  2. Fields created: Null.
  3. Field modified: Null.
  4. Outputs: (numeric [int] list: *NcharPlays*)

- ***get_Bodies(****nplays*[] = *NPlays)*: returns a character list containing the body of the selected plays inside the object. (returns character list)

  1. Inputs: (numeric [int] list: *ID*) or Null.
  2. Fields created: Null.
  3. Field modified: Null.
  4. Outputs: (character list)

- ***get_Speakers(****nplays*[] = *NPlays)*: returns a character list containing the name of the unique Speakers from each selected play inside the object. (returns character list)

  1. Inputs: (numeric [int] list: *ID*) or Null.
  2. Fields created: *CleanPlaysBody* (if non-existent), *SpeakerTextPlays* (if non-existent), *Speakers* (if non-existent).
  3. Field modified: *CleanedPlays*.
  4. Outputs: (character list: *Speakers*)

- ***get_NSpeakers(****nplays*[] = *NPlays)*: returns a numeric [int] list containing the number of the unique Speakers from each selected play inside the object. (returns numeric [int] list)

  1. Inputs: (numeric [int] list: *ID*) or Null.

2. Fields created: *CleanPlaysBody* (if non-existent), *SpeakerTextPlays* (if non-existent), *Speakers* (if non-existent).
3. Field modified: *CleanedPlays*.
4. Outputs: (numeric [int] list: *NSpeakers*)

- ***get_Words(****nplays*[] = *NPlays)*: returns a character list containing the words strings from each selected play inside the object. (returns character list)

1. Inputs: (numeric [int] list: *ID*) or Null.
2. Fields created: *CleanPlaysBody* (if non-existent), *SpeakerTextPlays* (if non-existent), *Speakers* (if non-existent), *Words* (if non-existent).
3. Field modified: *CleanedPlays*, *Words* (if existent).
4. Outputs: (character list: *Words*)

- ***get_NWords(****nplays*[] = *NPlays)*: returns a numeric [int] list containing the total number of the words from each selected play inside the object. (returns numeric [int] list)

1. Inputs: (numeric [int] list: *ID*) or Null.
2. Fields created: *CleanPlaysBody* (if non-existent), *SpeakerTextPlays* (if non-existent), *Speakers* (if non-existent), *Words* (if non-existent), *NWords* (if non-existent).
3. Field modified: *CleanedPlays*, *Words* (if existent), *NWords* (if existent).
4. Outputs: (numeric [int] list from: *NWords*)

- ***get_AVGWords(****nplays*[] = *NPlays)*: returns a numeric [float] list containing the average number of words per spoken chunk from each selected play inside the object. (returns numeric [float] list)

1. Inputs: (numeric [int] list: *ID*) or Null.
2. Fields created: *CleanPlaysBody* (if non-existent), *SpeakerTextPlays* (if non-existent), *Speakers* (if non-existent), *Words* (if non-existent), *NWords* (if non-existent), *NChunks*.
3. Field modified: *CleanedPlays*, *Words* (if existent), *NWords* (if existent), *NChunks* (if existent).
4. Outputs: (numeric [float] list from: *AVGWordsChunks*)

- ***get_Summary(****nplays*[] = *NPlays)*: returns a DataFrame object containing the current available statistics from each selected play inside the object. Hence, if fields such as *NWords* have not been initialized, the resulting DataFrame will not include them as columns. (returns DataFrame)

1. Inputs: (numeric [int] list: *ID*) or Null.
2. Fields created: *DataFrame* (if non-existing)
3. Field modified: *DataFrame* (if existing)
4. Outputs: (DataFrame with character and numeric fields: *SummaryTable*)

- ***get_BodyText(****nplays*[] = *NPlays)*: returns a character list containing the body of each selected play inside the object. (returns character list)

1. Inputs: (numeric list) or Null.
2. Fields created: Null.
3. Field modified: Null
4. Outputs: (character list from: *PreProcPlaysList* or *CleanPlaysBody* if existing)

- ***get_SpeakersText(****nplays*[] = *NPlays****)***: returns a character nested list containing the perfect match information between Speakers and text of each selected play inside the object. (returns character nested list)

  1. Inputs: (numeric [int] list: *ID*) or Null.
  2. Fields created: *CleanPlaysBody* (if non-existent), *SpeakerTextPlays* (if non-existent), *Speakers* (if non-existent).
  3. Field modified: *CleanedPlays*, *ObjectPlaysList*.
  4. Outputs: (character list from: *SpeakerTextPlays*)

- ***save_Plays(****filename, nplay*[] = *NPlays****)***: creates a file containing the full body of the selected plays. (void method)

  1. Inputs: (string character), (numeric [int] list: *ID*) or Null
  2. Fields created: Null.
  3. Field modified: Null.
  4. Outputs: Null. (void method, creates a file)

- ***del_Play(****nplays*[]***)***: deletes a subset of existing plays (by *ID* inside the object. (void method)

  1. Inputs: (numeric [int] list: *ID*)
  2. Fields created: None.
  3. Field modified: *ID*, *PreProcPlayList* (if existing), *ObjectPlaysList* (if existing), *CleanPlaysBody* (if existing), *SpeakerTextPlays* (if existing), *Titles*, *NActs*, *NScenes*, *NPlays*, *NYears*, *NSpeakers*, *NUnwords*, *NWords*, *Speakers*, *UniqueWords*, *Words*, *AVGWordsChunks*, *SummaryTable*, *NCharsPlays*, *CleanedPlay*.
  4. Outputs: Null (void method).

- ***add_Play(****Playobject*[]***)***: adds a new play from a *Play* Class object, calculating all relevant elements and updating a series of fields depending on their existence or not. (void method)

  1. Inputs: (object list)
  2. Fields created: None.
  3. Field modified: *ID*, *PreProcPlayList* (if existing), *ObjectPlaysList* (if existing), *CleanPlaysBody* (if existing), *SpeakerTextPlays* (if existing), *Titles*, *NActs*, *NScenes*, *NPlays*, *NYears*, *NSpeakers*, *NUnwords*, *NWords*, *Speakers*, *UniqueWords*, *Words*, *AVGWordsChunks*, *SummaryTable*, *NCharsPlays*, *CleanedPlay*.
  4. Outputs: Null (void method).

- ***print_Plays(****nplays*[] = *NPlays)*: prints out the selected plays in the main terminal screen for visualization purposes. (void method)

    1. Inputs: (numeric [int] list: *ID*) or Null
    2. Fields created: Null.
    3. Field modified: Null.
    4. Outputs: Null (void method)

- ***plot_TimeTrend(****Field, nplays*[] = *NPlays)*: display a plot of the relevant statistic vs Time (years) for the specific subset of plays given by the user. (void method)

    1. Inputs: (numeric list), (numeric [int] list: *ID*) or Null
    2. Fields created: Null.
    3. Field modified: Null.
    4. Outputs: Null (void method)

- ***clean_Plays(****nplays*[] = *NPlays)*: cleans the body of the given subset of plays. (void method)

    1. Inputs: (numeric [int] list: *ID*) or Null
    2. Fields created: *CleanPlaysBody* (if non-existing).
    3. Field modified: *CleanedPlay*, *ObjectPlaysList* (if existing).
    4. Outputs: Null (void method)

---

b.2) Class Plays: function, inputs, modifications and outputs.

As before, the methods defined inside this class are individual adaptations from the previously defined and described methods. Hence, methods like *print_Plays()* will be adapted to *print_Play()* without needing any numerical input since we are dealing with an individual play inside each object of this class. Same logic for the outputs: instead of returning lists, an individual value (numeric or character) will be returned. \

Important is to note that in this case the constructor method will explicitly require the raw text file for initializing the object. We will not allow the possibility of initialize this object by elements from another (a third) class.\

Finally, some methods will be no longer valid/useful such as *add_Play()* or *get_NPlays()* and can be omitted inside the *Plays* Class.