Text Manipulation

Statistical Computing, STA3005

Tuesday Jan 29, 2024

Last chapter: Dplyr, pipes, and more

`tidyverse` is a collection of packages for common data science tasks

- Tidyverse functionality is greatly enhanced using pipes (`%>%` operator)
- Pipes allow you to string together commands to get a flow of results

`dplyr` is a package for data wrangling, with several key verbs (functions)

- `filter()` : subset rows based on a condition
- `group_by()` : define groups of rows according to a condition
- `summarize()` : apply computations across groups of rows
- `arrange()` : order rows by value of a column
- `select()` : pick out given columns
- `mutate()` : create new columns
- `mutate_at()` : apply a function to given columns

`tidyr` is a package for manipulating the structure of data frames

- `pivot_longer()` : make "wide" data longer
- `pivot_wider()` : make "long" data wider

`ggplot2` graphics is comprised of data, layer, scale, coordinate, facet and theme, which are combined by `+`

- layer: `ggplot` for the coordinate system; `geom_point` for scatter or jitter plots; `geom_line` , `geom_bar` , `geom_boxplot` , `geom_histogram` for line charts, bar plots, boxplots and histogram, respectively
- scale: allow changing the color, size and type of points, lines, axis and legend
- coordinate: set a polar coordinate for pie chart `coord_polar` ; flip Cartesian coordinates `coord_flip`
- facet: create the same plot for different subsets
- theme: adjust display finer, like font size, background color, or no background layer (`theme_void()`)

Part I: *String basics*

What are strings?

We can easily distinguish characters and strings by their definitions:

- **Character:** a symbol in a written language, like letters, numerals, punctuation, space, etc.

- **String**: a sequence of characters bound together, such as a word, a sentence, or even a chapter

```
class("r")
```

```
## [1] "character"
```

```
class("STA3005")
```

```
## [1] "character"
```

Nevertheless, characters can be regarded as length-1 strings. Thus, both strings and characters belong to the `character` class. That's to say, they are regarded as the same thing in `R`.

Why do we care about strings?

- A lot of interesting data out there is in text format, like chatGPT!
- Webpages, emails, surveys, logs, search queries, etc.
- Even if you just care about numbers eventually, you'll need to understand how to get numbers from text

Whitespaces

Whitespaces are regarded as characters and can be included in strings:

- `" "` for space
- `"\n"` for newline
- `"\t"` for tab

```
str = "Dear Mr. Zhao,\n\nI have received your reply!\n\nSincerely,\n\nFangda"
str
```

```
## [1] "Dear Mr. Zhao,\n\nI have received your reply!\n\nSincerely,\n\nFangda"
```

A backslash `\` is the escape character in `R`, and `\n` and `\t` are called escape sequences. These escape sequences have special meanings.

Use `cat()` instead of `print()` to print strings to the console, displaying whitespaces properly

```
print(str)
```

```
## [1] "Dear Mr. Zhao,\n\nI have received your reply!\n\nSincerely,\n\nFangda"
```

```
cat(str)
```

```
## Dear Mr. Zhao,
##
## I have received your reply!
##
## Sincerely,
##
## Fangda
```

Vectors/matrices of strings

Like `numeric` and `logical`, `character` is a basic data type in `R`, so we can make vectors or matrices out of them. Just like we have done with numbers

```
str.vec = c("Statistical", "Computing", "is pretty good") # Collect 3 strings
str.vec # All elements of the vector
```

```
## [1] "Statistical"    "Computing"      "is pretty good"
```

```
str.vec[3] # The 3rd element
```

```
## [1] "is pretty good"
```

```
str.vec[-(1:2)] # All but the 1st and 2nd
```

```
## [1] "is pretty good"
```

```
str.mat = matrix("", 2, 3) # Build an empty 2 x 3 matrix
str.mat[1,] = str.vec # Fill the 1st row with str.vec
str.mat[2,1:2] = str.vec[1:2] # Fill the 2nd row, only entries 1 and 2, with
                              # those of str.vec
str.mat[2,3] = "is very important" # Fill the 2nd row, 3rd entry, with a new string
str.mat # All elements of the matrix
```

```
##      [,1]          [,2]        [,3]
## [1,] "Statistical" "Computing" "is pretty good"
## [2,] "Statistical" "Computing" "is very important"
```

```
t(str.mat) # Transpose of the matrix
```

```
##      [,1]             [,2]
## [1,] "Statistical"    "Statistical"
## [2,] "Computing"      "Computing"
## [3,] "is pretty good" "is very important"
```

Converting other data types to strings

Convert numeric or logical values into strings by `as.character()`

```
as.character(0.8)
```

```
## [1] "0.8"
```

```
as.character(0.8e+10)
```

```
## [1] "8e+09"
```

```
as.character(1:5)
```

```
## [1] "1" "2" "3" "4" "5"
```

```
as.character(TRUE)
```

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

Converting strings to other data types

Be careful! Depend on the contents of the given string

```r
as.numeric("0.5")
```

```
## [1] 0.5
```

```r
as.numeric("0.5 ")
```

```
## [1] 0.5
```

```r
as.numeric("0.5e-10") # in scientific notation
```

```
## [1] 5e-11
```

```r
as.numeric("Hi!")
```

```
## Warning: NAs introduced by coercion
```

```
## [1] NA
```

(若无法 coercion，则输出NA)

```r
as.logical("True")
```

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

```r
as.logical("TRU")
```

```
## [1] NA
```

Number of characters

Use `nchar()` to count the number of characters in a string

```r
nchar("coffee")
```

```
## [1] 6
```

```r
nchar("code monkey\n") # whitespaces are also counted
```

```
## [1] 12
```

- `\n` is counted as a single character instead of two.

```r
length("code monkey")
```

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

```r
length(c("coffee", "code monkey"))
```

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

```
nchar(c("coffee", "code monkey")) # Vectorization!
```

```
## [1]  6 11
```

Part II: *Substrings, splitting and combining strings*

Getting a substring

Use `substr()` to grab a subsequence of characters from a string, called a **substring**

```
phrase = "Give me a break"
substr(phrase, 1, 4)
```

```
## [1] "Give"
```

```
substr(phrase, nchar(phrase)-4, nchar(phrase))
```

(长度是5)

```
## [1] "break"
```

```
substr(phrase, nchar(phrase)+1, nchar(phrase)+10)
```

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

`substr()` vectorizes

Just like `nchar()`, and many other string functions

```
presidents = c("Clinton", "Bush", "Reagan", "Carter", "Ford")
# Grab the first 2 letters from each
substr(presidents, 1, 2)
```

```
## [1] "Cl" "Bu" "Re" "Ca" "Fo"
```

```
# Grab the first, 2nd, 3rd, etc.
substr(presidents, 1:5, 1:5)
```

⇒ substr(1,1), substr(2,2), substr(3,3) ……

```
## [1] "C" "u" "a" "t" ""
```

```
# Grab the first, first 2, first 3, etc.
substr(presidents, 1, 1:5)
```

⇒ substr(1,1), substr(1,2), substr(1,3) ……

```
## [1] "C"     "Bu"    "Rea"   "Cart" "Ford"
```

```
# Grab the last 2 letters from each
substr(presidents, nchar(presidents)-1, nchar(presidents))
```

```
## [1] "on" "sh" "an" "er" "rd"
```

```
## [1] on  sh  an  er  rd
```

Replace a substring

Can also use `substr()` to replace a character, or a substring

```
phrase
```

```
## [1] "Give me a break"
```

```
substr(phrase, 1, 1) = "L"
phrase # "G" changed to "L"
```

```
## [1] "Live me a break"
```

```
substr(phrase, 1000, 1001) = "R"
phrase # Nothing happened
```

```
## [1] "Live me a break"
```

```
substr(phrase, 1, 4) = "Show"
phrase # "Live" changed to "Show"
```

```
## [1] "Show me a break"
```

Splitting a string

Use the `strsplit()` function to split based on a keyword passed by the `split` argument

```
ingredients = "chickpeas, tahini, olive oil, garlic, salt"
split.obj = strsplit(ingredients, split=",")
split.obj
```

```
## [[1]]
## [1] "chickpeas"  " tahini"    " olive oil" " garlic"    " salt"
```

```
class(split.obj)
```

```
## [1] "list"
```

```
length(split.obj)
```

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

- The output is actually a list with just one element—a vector of strings.

`strsplit()` vectorizes

Just like `nchar()`, `substr()`, and the many others

```r
great.profs = "Nugent, Genovese, Greenhouse, Seltman, Shalizi, Ventura"
favorite.cats = "tiger, leopard, jaguar, lion"
split.list = strsplit(c(ingredients, great.profs, favorite.cats), split=",")
split.list
```

```
## [[1]]
## [1] "chickpeas"  " tahini"    " olive oil" " garlic"    " salt"
##
## [[2]]
## [1] "Nugent"      " Genovese"   " Greenhouse" " Seltman"    " Shalizi"
## [6] " Ventura"
##
## [[3]]
## [1] "tiger"    " leopard" " jaguar"  " lion"
```

- Returned object is a list with 3 elements
- Each one a vector of strings, having lengths 5, 6, and 4
- Do you see why `strsplit()` needs to return a list now?

Splitting character-by-character

Finest splitting you can do is character-by-character: use `strsplit()` with `split=""`

```r
split.chars = strsplit(ingredients, split="")[[1]]
split.chars
```

```
##  [1] "c" "h" "i" "c" "k" "p" "e" "a" "s" "," " " "t" "a" "h" "i" "n" "i" "," " "
## [20] "o" "l" "i" "v" "e" " " "o" "i" "l" "," " " "g" "a" "r" "l" "i" "c" "," " "
## [39] "s" "a" "l" "t"
```

```r
length(split.chars)
```

```
## [1] 42
```

```r
nchar(ingredients) # Matches the previous count
```

```
## [1] 42
```

Combining strings

Use the `paste()` function to join two (or more) strings into one, separated by a keyword `sep`

```r
paste("Spider", "Man") # Default is to separate by " "
```

```
## [1] "Spider Man"
```

```r
paste("Spider", "Man", sep="-")
```

```
## [1] "Spider-Man"
```

```r
paste("Spider", "Man", "does whatever", sep=", ")
```

```
## [1] "Spider, Man, does whatever"
```

`paste()` vectorizes

Just like `nchar()`, `substr()`, `strsplit()`, etc. Seeing a theme yet?

```
presidents
```

```
## [1] "Clinton" "Bush"    "Reagan"  "Carter"  "Ford"
```

```
paste(presidents, c("D", "R", "R", "D", "R"))
```

```
## [1] "Clinton D" "Bush R"    "Reagan R"  "Carter D"  "Ford R"
```

```
paste(presidents, c("D", "R")) # Notice the recycling (not historically accurate!)
```

```
## [1] "Clinton D" "Bush R"    "Reagan D"  "Carter R"  "Ford D"
```

```
paste(presidents, " (", 42:38, ")", sep="")
```

```
## [1] "Clinton (42)" "Bush (41)"    "Reagan (40)"  "Carter (39)"  "Ford (38)"
```

Condensing a vector of strings

`paste()` with the `collapse` argument can compress a vector of strings into one big string

```
presidents
```

```
## [1] "Clinton" "Bush"    "Reagan"  "Carter"  "Ford"
```

```
paste(presidents, collapse="; ")
```

```
## [1] "Clinton; Bush; Reagan; Carter; Ford"
```

```
paste(presidents, " (", 42:38, ")", sep="", collapse="; ")
```

```
## [1] "Clinton (42); Bush (41); Reagan (40); Carter (39); Ford (38)"
```

```
paste(presidents, " (", c("D", "R", "R", "D", "R"), 42:38, ")", sep="", collapse="; ")
```

```
## [1] "Clinton (D42); Bush (R41); Reagan (R40); Carter (D39); Ford (R38)"
```

```
paste(presidents, collapse=NULL) # No condensing, the default
```

```
## [1] "Clinton" "Bush"    "Reagan"  "Carter"  "Ford"
```

Part III: *Matching and substituting as a pattern*

Search for matching a pattern

Use `grep()` to return the indices in a given character vector `x` matching a **regular expression** passed by the `pattern` argument

```
grep(pattern = "^C", presidents)
```

```
## [1] 1 4
```

- The regular expression `"^C"` means that the string starts with `C`. We will discuss more about regular expressions a bit later, simply thought of as a special string.

```
grep(pattern = "^C", presidents, value = TRUE)
```

```
## [1] "Clinton" "Carter"
```

- The argument `value` controls outputting indices (`value = FALSE`) or values (`value = TRUE`)

Substitute a pattern

We have learned to replace a substring by `substr()`. However, that requires the locations of substring to be replaced. Use `gsub` to substitute the strings matched to a regular expression by another string provided in `replacement`.

```
# "C"s at the first place are replaced by "X"
gsub(pattern = "^C", replacement = "X", presidents)
```

```
## [1] "Xlinton" "Bush"    "Reagan"  "Xarter"  "Ford"
```

```
# "C"s at the last place are replaced by "X"
gsub(pattern = "n$", replacement = "X", presidents)
```

```
## [1] "ClintoX" "Bush"    "ReagaX"  "Carter"  "Ford"
```

- `pattern` needs a regular expression, while `replacement` requires a general string.
- Both `grep` and `gsub` vecterize, similar to `paste`, `strsplit`, etc.

Regular Expressions

A regular expression in `R` is a string that describes a certain pattern found in a text. When used in R regular expressions, a few characters that have a special meaning.

Define an example character data frame

```
df_str <- data.frame(Plot.ID = c("1_A","1_A","1_B","2_A","2_A"),
                     Species = c("GF","WS","F","GF","GF var. Bupkiss"),
                     Dbh = c("18.8","NA","20.0","25.8","24"),
                     Color = c("gray","grey","groy","red", "blue"),
                     Mix = c(5, "dogs", "sit", "on", "grass"))
```

Structures

- `()` : grouping
- `[]` : any one of them
- `.` : any single character
- `|` : or

```
grep("gr(a|e)y", df_str$Color, value = TRUE)
```

```
## [1] "gray" "grey"
```

```
grep("gray|grey", df_str$Color, value = TRUE)
```

```
## [1] "gray" "grey"
```

```
grep("gr[ae]y", df_str$Color, value = TRUE)
```

```
## [1] "gray" "grey"
```

```
grep("gr[a-z]y", df_str$Color, value = TRUE)
```

注意大小写

```
## [1] "gray" "grey" "groy"
```

```
grep("gr..", df_str$Color, value = TRUE)
```

```
## [1] "gray" "grey" "groy"
```

Anchors

`^c` and `c$` denotes starting and ending with a character `c`, respectively

```
grep("^F", df_str$Species, value = TRUE)
```

```
## [1] "F"
```

```
grep("F$", df_str$Species, value = TRUE)
```

```
## [1] "GF" "F"  "GF"
```

```
grep("^F$", df_str$Species, value = TRUE)
```

```
## [1] "F"
```

Character Classes

Built-in character classes:

- `[:alpha:]` is equivalent to `A-Za-z`
- `[:digit:]` is equivalent to `0-9`
- `[:space:]` includes all possible whitespaces, like `\n` and `\t`

- `[:punct:]` includes all punctuation marks, like `,` and `:`

```
grep("^[[:alpha:]]", df_str$Mix, value = TRUE)
```

end with any of A-Z/a-z

```
## [1] "dogs"  "sit"   "on"    "grass"
```

```
grep("^[[:digit:]]", df_str$Mix, value = TRUE)
```

```
## [1] "5"
```

Why do we have double brackets?

- The outside brackets means to match any one of charterers in the character class

Quantifiers

- `c?` : zero or one 0/1
- `c*` : zero or more ≥0
- `c+` : one or more ≥1
- `c{n}` : exactly n times =n

```
# Zero or one character between two Fs
grep("^F.?F$", c("F","FG","GF","FF","FaFa","FabcF"), value = TRUE)
```

```
## [1] "FF"
```

```
# Zero or more characters between two Fs
grep("^F.*F$", c("F","FG","GF","FF","FaFa","FabcF"), value = TRUE)
```

```
## [1] "FF"    "FabcF"
```

```
# At least one characters between two Fs
grep("^F.+F$", c("F","FG","GF","FF","FaFa","FabcF"), value = TRUE)
```

```
## [1] "FabcF"
```

```
# Exactly three characters between two Fs
grep("^F.{3}F$", c("F","FG","GF","FF","FaFa","FabcF"), value = TRUE)
```

```
## [1] "FabcF"
```

Escape character

- For general strings, single backslash `\` is the escape character (转义字符)
- Characters escape from the literal meanings, like `\n` for newlines
- For regular expressions, double backslash is the escape character
- Specail characters escape from their special meanings, like `\\.` matching `.` in strings

```
grep(".", df_str$Dbh, value = TRUE)
```

含有任意 char

```
## [1] "18.8" "NA"   "20.0" "25.8" "24"
```

```
grep("\\.", df_str$Dbh, value = TRUE)
```

含有"."

```
## [1] "18.8" "20.0" "25.8"
```

```
gsub(pattern = ".", replacement = "*", df_str$Dbh)
```

```
## [1] "****" "**"   "****" "****" "**"
```

```
gsub(pattern = "\\.", replacement = "*", df_str$Dbh)
```

```
## [1] "18*8" "NA"   "20*0" "25*8" "24"
```

```
gsub(pattern = "\\.", replacement = "..", df_str$Dbh)
```

```
## [1] "18..8" "NA"    "20..0" "25..8" "24"
```

- Again, `pattern` requires a regular expression, while `replacement` only passes a string.

In summary, special characters in R regular expressions include

- Structures: `[]`, `()`, `|`, `.`
- Anchors: `^` and `$`
- Character Classes: `[:alpha:]`, `[:digit:]`, `[:punct:]`
- Quantifiers: `?`, `*`, `+`, `{}`
- Escape character: `\\`

Utility of regular expressions

Ex1: Count how many words in the following sentence starting with `i`

```
sentence <- "Don't aim for success if you want it. Just do what you love and believe in, and

library(tidyverse)
```

```
## ── Attaching core tidyverse packages ──────────────────────── tidyverse 2.0.0 ──
## ✔ dplyr     1.1.3     ✔ readr     2.1.4
## ✔ forcats   1.0.0     ✔ stringr   1.5.0
## ✔ ggplot2   3.4.3     ✔ tibble    3.2.1
## ✔ lubridate 1.9.3     ✔ tidyr     1.3.0
## ✔ purrr     1.0.1
## ── Conflicts ────────────────────────────────────────── tidyverse_conflicts() ──
## ✖ dplyr::filter() masks stats::filter()
## ✖ dplyr::lag()    masks stats::lag()
## ℹ Use the ]8;;http://conflicted.r-lib.org/ conflicted package ]8;;  to force all conflict
```

```
sentence %>%
  strsplit(split = " ") %>%
  .[[1]] %>%
  grep(pattern = "^i",.) %>%
  length
```

```
## [1] 4
```

Ex2: How to check whether a series of email addresses are CUHK-SZ account?

```
emails <- c("songfangda@cuhk.edu.cn", "1350923@163.com","3423422@link.cuhk.edu.cn","3423422@

grep(pattern = ".*@(link.)?cuhk.edu.cn$", emails, value = TRUE)
```

需要转义字符

```
## [1] "songfangda@cuhk.edu.cn"   "3423422@link.cuhk.edu.cn"
## [3] "2342343@link1cuhk2edu3cn"
```

```
grep(pattern = ".*@(link\\.)?cuhk\\.edu\\.cn$", emails, value = TRUE)
```

```
## [1] "songfangda@cuhk.edu.cn"   "3423422@link.cuhk.edu.cn"
```

Part IV: *Reading in text, summarizing text*

Text from the outside

How to get text, from an external source, into R by using the `readLines()` function

```
king.lines = readLines("king.txt")
class(king.lines) # We have a string vector
```

```
## [1] "character"
```

```
length(king.lines) # Many lines (elements)!
```

```
## [1] 59
```

```
king.lines[1:3] # First 3 lines
```

```
## [1] "Five score years ago, a great American, in whose symbolic shadow we stand today, sig
## [2] ""
## [3] "But 100 years later, the Negro still is not free. One hundred years later, the life
```

This file stores Martin Luther King Jr.'s famous "I Have a Dream" speech at the March on Washington for Jobs and Freedom on August 28, 1963

Reconstitution

Fancy word, but all it means: make one long string, then split the words

```
king.text = paste(king.lines, collapse=" ")
king.words = strsplit(king.text, split=" ")[[1]]

# Sanity check
substr(king.text, 1, 150)
```

```
## [1] "Five score years ago, a great American, in whose symbolic shadow we stand today, sig
```

```
king.words[1:20]
```

```
##  [1] "Five"          "score"         "years"         "ago,"
##  [5] "a"             "great"         "American,"     "in"
##  [9] "whose"         "symbolic"      "shadow"        "we"
## [13] "stand"         "today,"        "signed"        "the"
## [17] "Emancipation"  "Proclamation." "This"          "momentous"
```

Counting words

Our most basic tool for summarizing text: **word counts**, retrieved using `table()`

```
king.wordtab = table(king.words)
class(king.wordtab)
```

```
## [1] "table"
```

```
length(king.wordtab)
```

```
## [1] 622
```

```
king.wordtab[1:10]
```

```
## king.words
##          'tis      - ...the  ...to    100   1963      a   able  Again
##     29      1      2      1      1      1      1     37      8      1
```

- Finally, we obtain alphabetically sorted unique words, and their counts = number of appearances

The names are words, the entries are counts

The output is actually a vector of numbers, and the words are the names of the vector

```
king.wordtab[1:5]
```

```
## king.words
##          'tis      - ...the  ...to
##     29      1      2      1      1
```

```
king.wordtab[2] == 2
```

```
##  'tis
## FALSE
```

```
names(king.wordtab)[2] == "-"
```

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

So with named indexing, we can now use this to look up whatever words we want

```
king.wordtab["dream"]
```

```
## dream
##     9
```

```
king.wordtab["freedom"]
```

```
## freedom
##      18
```

```
king.wordtab["equality"] # NA means King never mentioned equality
```

```
## <NA>
##   NA
```

Does `equality` never appear in the paragraph? No!

```
grep(pattern = "equality", names(king.wordtab), value = TRUE)
```

```
## [1] "equality."
```

```
grep(pattern = "York", names(king.wordtab), value = TRUE)
```

```
## [1] "York"  "York."
```

- Punctuation matters: e.g., “York” and “York.” are treated as separate words, not ideal.

Therefore, we exclude all punctuation by `gsub()`

(把标点替换为" ")

```
king.words.nopun <- gsub(pattern = "[[:punct:]]", "", king.words)
king.wordtab = table(king.words.nopun)
```

Most frequent words

Let’s sort in decreasing order, to get the most frequent words

```
king.wordtab.sorted = sort(king.wordtab, decreasing=TRUE)
length(king.wordtab.sorted)
```

```
## [1] 560
```

```
head(king.wordtab.sorted, 10) # First 20
```

```
## king.words.nopun
##    of  the   to  and    a   be      will that   is
##    98   98   58   40   37   32   31   25   24   23
```

```
tail(king.wordtab.sorted, 10) # Last 20
```

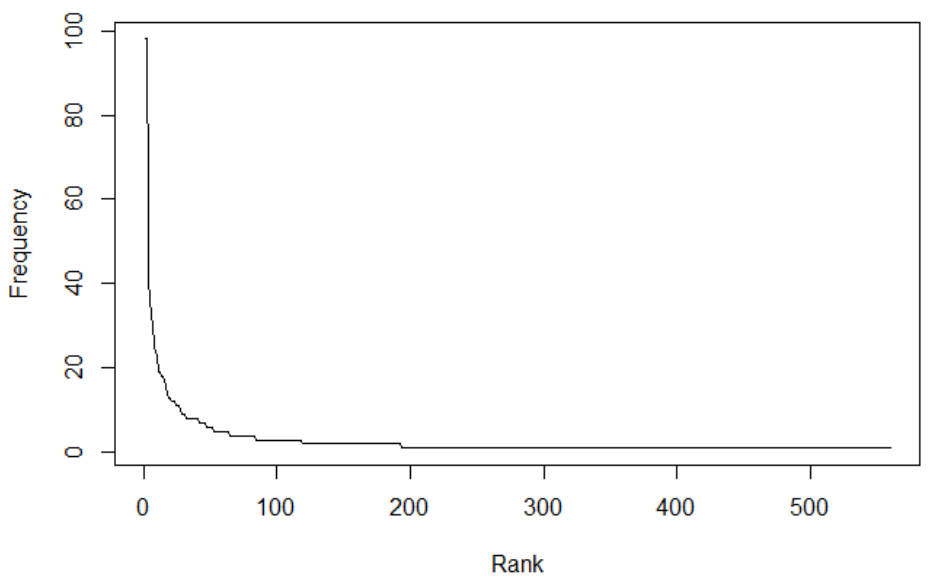
```
## king.words.nopun
```

```
## king.words.nopun
## whirlwinds     whites      whose      winds  withering   wrongful      wrote
##          1          1          1          1          1          1          1
##        yes        You       your
##          1          1          1
```

Visualizing frequencies

Let's use a plot to visualize frequencies

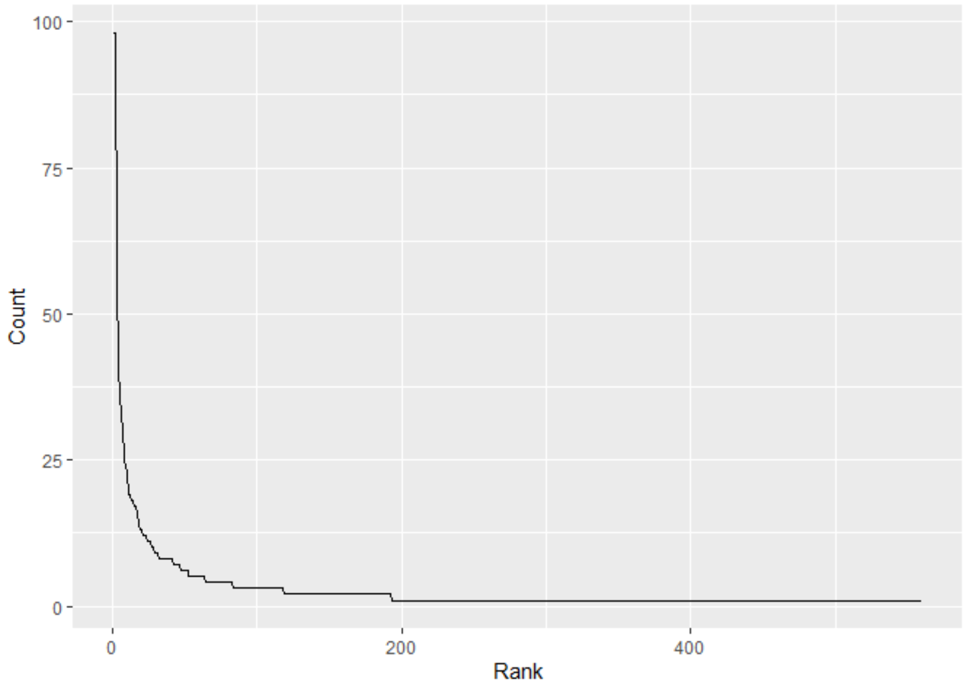
```
nw = length(king.wordtab.sorted)
plot(1:nw, as.numeric(king.wordtab.sorted), type="l",
     xlab="Rank", ylab="Frequency")
```



By ggplot2 ,

```
king.df <- data.frame(Word = names(king.wordtab),
                      Count = as.vector(king.wordtab))

king.df %>%
  arrange(desc(Count)) %>%
  mutate(Rank = 1:nrow(.)) %>%
  ggplot(aes(x = Rank, y = Count)) +
  geom_line()
```



A pretty drastic looking trend! It looks as if $\text{Frequency} \propto (1/\text{Rank})^a$ for some constant $a > 0$

Zipf's law

This phenomenon, that frequency tends to be inversely proportional to a power of rank, is called **Zipf's law**

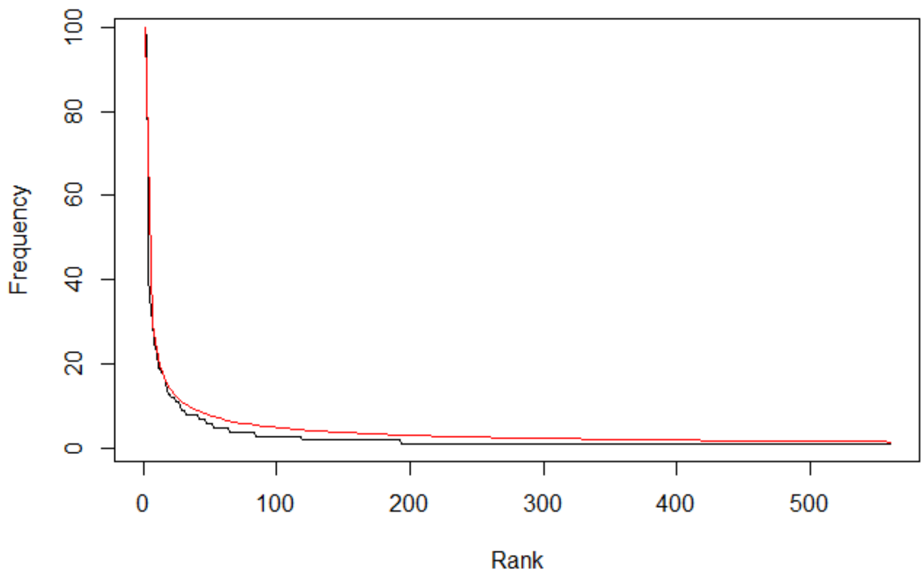
For our data, Zipf's law approximately holds, with $\text{Frequency} \approx C(1/\text{Rank})^a$ for $C = 100$ and $a = 0.65$

```
C = 100; a = 0.65
king.wordtab.zipf = C*(1/1:nw)^a
cbind(king.wordtab.sorted[1:8], king.wordtab.zipf[1:8])
```

```
##       [,1]      [,2]
## of      98 100.00000
## the     98  63.72803
## to      58  48.96336
## and     40  40.61262
## a       37  35.12930
## be      32  31.20338
##         31  28.22840
## will    25  25.88162
```

Not perfect, but not bad. We can also plot the original sorted word counts, and those estimated by our formula law on top

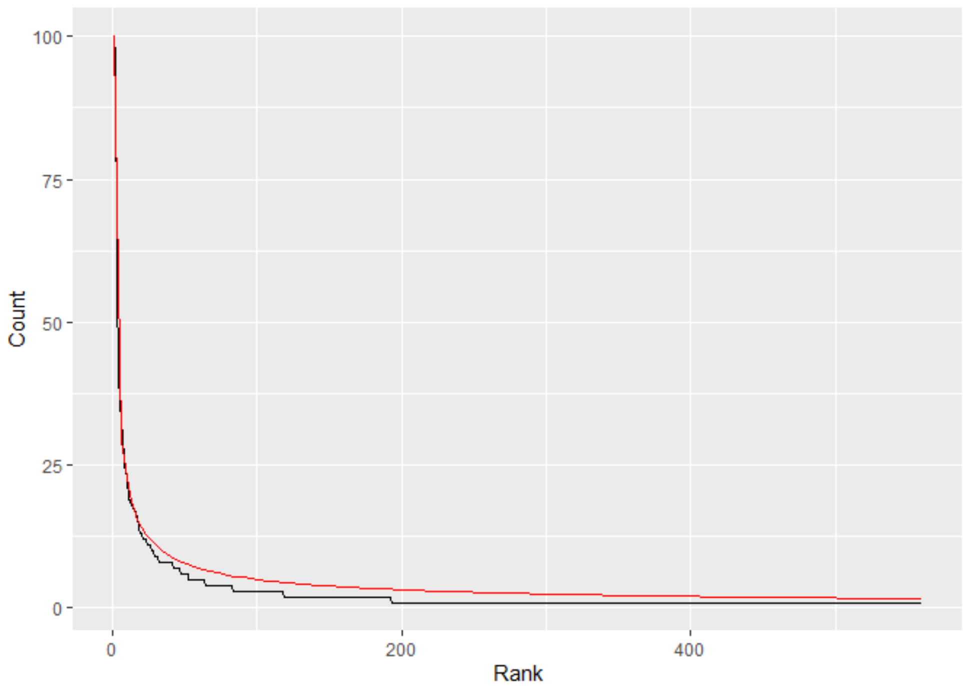
```
plot(1:nw, as.numeric(king.wordtab.sorted), type="l",
     xlab="Rank", ylab="Frequency")
curve(C*(1/x)^a, from=1, to=nw, col="red", add=TRUE)
```



By `ggplot2`,

```
king.df <- data.frame(Word = names(king.wordtab),
                      Count = as.vector(king.wordtab))

king.df %>%
  arrange(desc(Count)) %>%
  mutate(Rank = 1:nrow(.)) %>%
  ggplot(aes(x = Rank, y = Count)) +
  geom_line() +
  geom_function(fun = function(x){C*(1/x)^a}, color = "red", xlim=c(1, nrow(king.df)))
```



We'll learn more about basic plotting tools in detail in the next chapter

Summary

- Strings are sequences of characters bound together
- Text data occurs frequently "in the wild", so you should learn how to deal with it!
- `nchar()`, `substr()`: functions for substring extractions and replacements
- `strsplit()`, `paste()`: functions for splitting and combining strings

- `grep()`, `gsub()`: functions for matching and substituting for a given regular expression
- A regular expression is a string that describe a certain text pattern. The pattern is expressed by the combination of general and special characters
- Reconstitution: take lines of text, combine into one long string, then split to get the words
- `table()`: function to get word counts, useful way of summarizing text data
- Zipf's law: word frequency tends to be inversely proportional to (a power of) rank