Basic text processing and corpus methods

Computational Linguistics (LING 455)

Rutgers University September 17, 2021

Strings are lists of characters

```
*Main> 'L':('i':('n':('g':('u':('i':('s':('t':[]))))))) 
"Linguist"
```

What are `Char`acters?

There are many, many characters:

- Unicode `'x'`, Σ, ...
- Spaces, punctuation, other symbols `' '`, `.`, `@`
- Newlines: `\n`
- Emoji, characters from non-Latin writing systems

```
Prelude> putStrLn "I\n am\n\n  taking\n      LING\n455"
I
 am

  taking
      LING
455
```

In terms of types, what are…

- Words? `String`? `[String]`?
- Sentences? `String`? `[String]`?
- Texts? `String`? `[String]`? `[[String]]`?

What if we wish to include information relevant for linguistic analysis (e.g., parts of speech, constituency grouping, etc)?

- How does this influence the data structures we will use?

`heads` and `tails`

A list is either `[]` or has the shape `x:xs`. Haskell pre-defines a `head` function for accessing `x` and a `tail` function for accessing `xs`:

```
*Main> head "Linguistics" -- head :: [a] -> a
'L'
```

```
*Main> tail "Linguistics" -- tail :: [a] -> [a]
"inguistics"
```

Neither of these functions works on the empty list:

```
*Main> head []
*** Exception: Prelude.head: empty list
```

```haskell
whatsThisDo1 :: [a] -> [a]
whatsThisDo1 []     = []
whatsThisDo1 (x:xs) = x : whatsThisDo1 xs
```

```haskell
whatsThisDo1 :: [a] -> [a]
whatsThisDo1 []     = []
whatsThisDo1 (x:xs) = x : whatsThisDo1 xs
```

```haskell
whatsThisDo2 :: [a] -> [a]
whatsThisDo2 [] = []
whatsThisDo2 (x:xs) = whatsThisDo2 xs ++ [x]
```

```haskell
whatsThisDo1 :: [a] -> [a]
whatsThisDo1 []     = []
whatsThisDo1 (x:xs) = x : whatsThisDo1 xs
```

```haskell
whatsThisDo2 :: [a] -> [a]
whatsThisDo2 [] = []
whatsThisDo2 (x:xs) = whatsThisDo2 xs ++ [x]
```

```haskell
isPalindrome :: String -> Bool
isPalindrome xs = xs == whatsThisDo2 xs
```

Important operations on lists: `filter` and `map`

```
*Main> filter odd [1..10]
[1,3,5,7,9]
```

```
*Main> map (\x -> x*2) [1..10]
[2,4,6,8,10,12,14,16,18,20]
```

These calculations can also be carried out using list comprehensions.

```
*Main> [ n | n <- [1..10], odd n ]
[1,3,5,7,9]
```

```
*Main> [ (\x -> x*2) n | n <- [1..10] ] -- or just n*2!
[2,4,6,8,10,12,14,16,18,20]
```

Higher-order functions

It is revealing to examine the types of `filter` and `map`:

```
*Main> :t filter
filter :: (a -> Bool) -> [a] -> [a]
-- takes a predicate of a's and a list of a's and
-- returns a list of a's satisfying the predicate
```

```
*Main> :t map
map :: (a -> b) -> [a] -> [b]
-- takes a recipe for turning a's to b's and a list
-- of a's, turning each of them into b, per the recipe
```

These functions are **higher-order**. They want a function as an input!

Practice: defining `filter` for ourselves

```haskell
-- using recursion
myFilter1 :: (a -> Bool) -> [a] -> [a]
myFilter1 f []     = []
myFilter1 f (x:xs) = if f x
                     then x : (myFilter1 f xs)
                     else myFilter1 f xs
```

```haskell
-- using a list comprehension
myFilter2 :: (a -> Bool) -> [a] -> [a]
myFilter2 f [] = []
myFilter2 f xs = [ x | x <- xs, f x ]
```

Practice: defining `map` for ourselves

```haskell
-- using recursion
myMap1 :: (a -> b) -> [a] -> [b]
myMap1 f []     = []
myMap1 f (x:xs) = f x : (myMap1 f xs)
```

```haskell
-- using a list comprehension
myMap2 :: (a -> b) -> [a] -> [b]
myMap2 f [] = []
myMap2 f xs = [ f x | x <- xs ]
```

A commonality

Both `filter` and `map` have in essence the same logic, traveling through a collection and applying some operation to its elements.

Many types of collections can be traversed in this way. We will see many examples of this pattern in this course.

The Data.List library

Data.List package comes with the Haskell Platform (link)

```
import Data.List
```

You can import in two different ways, depending on your needs:

- At the **very top** of a hs file (see W3.hs)
- At any time in a ghci session

Data.List is a library with many very useful functions:

```
*Main> nub [1,2,3,2,1,3,2,3,2,2,1,1]
[1,2,3]
*Main Data.List> sort ["the","quick","brown","fox"]
["brown","fox","quick","the"]
```

Working with texts/corpora: cleaning text

Loading a text

Loading `great-expectations.txt` as a `String` we'll call `raw`:

```
*Main> raw <- readFile "aux/great-expectations.txt"
```

Why '`raw`'? Well it is pretty raw!

```
*Main> take 235 raw
"\65279Great Expectations\n\nChapter I\n\nMy father's
family name being Pirrip, and my Christian name Philip,
my\ninfant tongue could make of both names nothing longer
or more explicit\nthan Pip. So, I called myself Pip, and
came to be called Pip."
```

Poking around

`great-expectations.txt` is a `String` $\approx 10^6$ characters long.

```
*Main> length raw
993790
```

Around 17,000 ($\approx 1.7\%$) are just commas!

```
*Main> justCommas = filter (== ',') raw
*Main> length justCommas
17050
```

Cute trick: in Haskell `(== ',')` is the same as `(\x -> x == ',')`! This works for any infix operator, on both the left and right: `(4 +)` is the same as `(\i -> 4 + i)`. (These are known as 'sections'.)

Tokenization

Tokenization is the process of breaking a completely unstructured `String` into chunks, most commonly **words**.

Some cases to think about that will likely cause unintended results:

- Periods and commas: `river, within, as the river wound,`
- Apostrophes: `my father's`, `Darn me if I couldn't`
- Hyphens: `sister,--Mrs. Joe`
- Capitalization: `A fearful man`, `with a great iron`

Before tokenizing, we need to clean the file a bit so these artifacts don't get in the way. To say nothing of harder cases in other contexts:

```
I told him to go <a href="https://google.com">here</a>.
```

Data.Char: a useful library for working with text (link)

```haskell
import Data.Char -- at the top of W3.hs
```

```
*Main> isLetter 'C'
True
*Main> isLetter '.'
False
```

```
*Main> isSpace ' '
True
*Main> isSpace '\n'
True
```

```
*Main> toLower 'M'
'm'
```

Some actions for cleaning text

```haskell
-- replace dashes with spaces
repDash :: String -> String
repDash xs = map (\c -> if c == '-' then ' ' else c) xs
```

```haskell
-- delete punctuation
delPunct :: String -> String
delPunct xs = filter (\x -> isLetter x || isSpace x) xs
```

```haskell
-- make all lowercase
allLower :: String -> String
allLower xs = map toLower xs
```

Chaining it all together

```
-- cleaning is doing one after the other
clean :: String -> String
clean xs = allLower (delPunct (repDash xs))
-- first replace dashes with spaces
-- then delete punctuation
-- finally, make all characters lowercase
```

Notice that we chain these operations in particular order.

- `delPunct` applies after `repDash`, else `sister,--Mrs. Joe` becomes `sisterMrs Joe` not `sister  Mrs Joe`.
- Sometimes order doesn't make a difference. (When?)

Function composition

Chaining operations is also known as **composition**.

Haskell has a concise way to write the composition of `f` and `g`: `f . g` (pronounced: "do `f` after `g`") is defined as `\xs -> f (g xs)`.

```
cleanComp :: String -> String
cleanComp = allLower . delPunct . repDash
--                  3rd       (2nd         1st    )
```

This is equivalent to `clean` (exercise: check!). It is also more concise: it doesn't mention `xs`, and parens are unnecessary:

- Composition is **associative**: `f . (g . h) == (f . g) . h`

Taking stock

```
*Main> cleaned = clean raw
*Main> length raw - length cleaned
40194
*Main> writeFile "aux/cleaned.txt" (clean raw)
```

```
great expectations

chapter i

my fathers family name being pirrip and my christian name philip my
infant tongue could make of both names nothing longer or more explicit
than pip so i called myself pip and came to be called pip

i give pirrip as my fathers family name on the authority of his
tombstone and my sister  mrs joe gargery who married the blacksmith
as i never saw my father or my mother and never saw any likeness
of either of them for their days were long before the days of
```

Taking stock (cont)

This representation of the text is still unstructured (not tokenized). And our clean-job is far from perfect (e.g., `chapter i` vs `i give`)!

- But as the text is quite long, it is reasonable to think that these imperfections are statistically not so meaningful.

Our strategy has been **lossy** with respect to some information we might reasonably want. Our sentences boundaries are gone forever.

- (Well, they're gone in `cleaned`. But we still have `raw`!)
- For now, we focus on **words**, and so sentence breaks are not such important information for us to have.
- In any case, finding a sentence boundary is quite nontrivial to do automatically. What are some issues you see?

`IO` in Haskell

The parts of a hs file that interact with the outside world have values of type `IO a`. `IO` stands for input/output:

```haskell
main :: IO ()
main = do
  raw <- readFile "aux/great-expectations.txt"
  let cleaned = clean raw -- needs `let`, idk why!
  writeFile "aux/cleaned.txt" cleaned
```

This so-called `do`-block can reference stuff defined elsewhere in your file, like `clean`. But *the reverse is not true!* If you try to reference `raw` outside the `do`-block, you will get an error.[1]

[1] The reasons aren't really our concern. But the idea is that Haskell distinguishes **pure** data — timeless functions that always do the same thing no matter what — and **impure** data, which behaves in different ways, depending on how the world looks.

`IO` in Haskell (cont)

There's one sort of exception, `ghci`:

```
*Main> raw <- readFile "aux/great-expectations.txt"
*Main> length raw
993790
```

This isn't really an exception, actually. When you're inside a `ghci` session, it's like you're in a big `do`-block.[2]

[2] Except in `ghci` you don't need to use `let` to define a new variable, though you can.

Tokenizing

`words`

Around 20,000 `Char`s in `clean raw` are `\n`, i.e., artifacts of the specific `txt` file that I downloaded, and generally not meaningful:

```
*Main> justBreaks = filter (== '\n') (clean raw)
*Main> length justBreaks
20008
```

Both `' '` and `\n` are considered types of spaces, and `Prelude` gives us a helpful function for breaking a `String` at its spaces:

```
*Main> :t words
words :: String -> [String]
*Main> words "I\n am\n\n  taking\n      LING\n455"
["I","am","taking","LING","455"]
```

Our tokenized text

```
tokenize :: String -> [String]
tokenize = words
```

And with that, we have our tokenized text, type `[String]`:

```
*Main> length (tokenize raw)
186591
```

This wordcount closely matches (within 2%) other estimates online.

Avg word length

Can we find the **average word length** in *Great Expectations*? (This is often considered a proxy for the 'reading level' or difficulty of a text.)

```
allLengths :: [String] -> [Int]
allLengths xs = map length xs
```

The average of a set s is given by $\frac{\sum s}{|s|}$. Or for a list `xs`: $\frac{\texttt{sum xs}}{\texttt{length xs}}$.

Avg word length

Can we find the **average word length** in *Great Expectations*? (This is often considered a proxy for the 'reading level' or difficulty of a text.)

```
allLengths :: [String] -> [Int]
allLengths xs = map length xs
```

The average of a set s is given by $\frac{\Sigma s}{|s|}$. Or for a list `xs`: $\frac{\texttt{sum xs}}{\texttt{length xs}}$.

```
-- firstTry xs = sum (allLengths xs) / length xs
-- won't work:
-- <interactive>:14:15: error:
--     • Could not deduce (Fractional Int) arising from
--       a use of '/'
--       from the context: Foldable t
```

`fromIntegral`

What went wrong? The `/` division operator needs `Fractional` numbers. And `Int`s aren't `Fractional` because they are **whole**.

Fortunately, it is straightforward to massage an `Int`:

```
*Main> x = 2 :: Int; y = 3 :: Int
*Main> x / y
<interactive>:19:1: error:
```

```
*Main> x' = fromIntegral x; y' = fromIntegral y
*Main> x' / y'
0.6666666666666666
```

Putting it all together

```haskell
ezDiv :: Int -> Int -> Double -- a floating point number
ezDiv m n = fromIntegral m / fromIntegral n

avgWordLength :: [String] -> Double
avgWordLength xs =
  let numerator   = sum (allLengths xs)
      denominator = length xs in
    ezDiv numerator denominator
```

```
*Main> avgWordLength (tokenize raw)
4.081836744537518
```

How many *distinct* words?

Suppose we wanted to find out how many distinct words in our text. We could use `nub` from `Data.List`. This works, but `nub` is slooow!

```
*Main> tokens = tokenize raw -- 186,591 words
*Main> :set +s -- this will time how long it takes
*Main> length (nub tokens)
10896
(19.92 secs, 536,329,824 bytes) -- way too long to wait!
```

The problem is that `nub` requires on the order of n^2 steps to work for a list of length n, and there's no way around this in general. For a list with 186,591 things, this is about 35 *billion* steps.

`sort` then `group`

Your reading talks about how using the right data structure, a `Set`, can help remedy these performance issues. Another possibility is to take advantage of the fact that words are *sortable* things.

Sorting can be efficient, and it puts like things next to each other:

```
*Main> sort ["i", "think", "i", "am", "tired"]
["am","i","i","think","tired"]
```

Once we have a `sorted` list, we can call `group`. And voila, we have grouped repeated words together!

```
*Main> group (sort ["i", "think", "i", "am", "tired"])
[["am"],["i","i"],["think"],["tired"]] -- 4 unique words
```

Benchmarking it

```haskell
import qualified Data.Set as Set
-- ...
```

```haskell
getTypes :: [String] -> [[String]]
getTypes = group . sort
```

```
*Main> Set.size (Set.fromList tokens)
10896
(0.11 secs, 76,311,664 bytes)
```

```
*Main> let types = getTypes tokens
*Main> length types
10896
(0.36 secs, 256,314,536 bytes) -- not bad!
```

Statistical properties of word **types**

We have our types and tokens

Now we know some things about the word **types**, along with the word **tokens** in our text.

```
*Main> length tokens
186591
*Main> length (getTypes tokens)
10896
```

We can begin to extract, and compute statistical information about word types. For example, how common are various word types? Do any regularities emerge about the relative frequencies of word types?

Word type frequencies

Say we wanted to find the most common words in our text. How would we go about it? Remember: `types` is actually a list where each element is itself a list, containing every occurrence of some word.

- The `head` element of `types` is 4,048 `"a"`s! LOL
- The next element is just one `"aback"`. :(

```
addCount :: [String] -> (Int, String)
addCount xs = (length xs, head xs)
```

If we `map` this over `types`, we pair each word type with how often it occurrred. So the first element will be `(4048, "a")`.

- Note the use of `head`: we don't need to lug around all 4048 `"a"`s, since we're already lugging around `4048`!

Trying it out

Our types are all paired with their raw counts. If we `sort` once more, these pairs will be now sorted *by that number!*—that is, ranked!

```
*Main> withCounts = map addCount types
*Main> sortedByCount = sort withCounts -- ascending
```

What's the most common word in *Great Expectations*?

Trying it out

Our types are all paired with their raw counts. If we `sort` once more, these pairs will be now sorted *by that number!*—that is, ranked!

```
*Main> withCounts = map addCount types
*Main> sortedByCount = sort withCounts -- ascending
```

What's the most common word in *Great Expectations*?

```
*Main> last sortedByCount
(8145, "the")
```

Proportional frequencies

We have sorted our tokens into types, counted how many instances of each type occur in our text, and ranked the words on that basis.

But *I* want to know more: what **proportion** of the words in the text does each of these types represent?

- A famous regularity in texts is known as **Zipf's Law**. In its simplest form: the 100th most common word in a text occurs 10x as often as the 1000th (and so on, for all such comparisons).
- This is a bit surprising: it could have been that all words were equally frequent, or that frequency decreased less drastically.

Does Zipf's Law hold for our text?

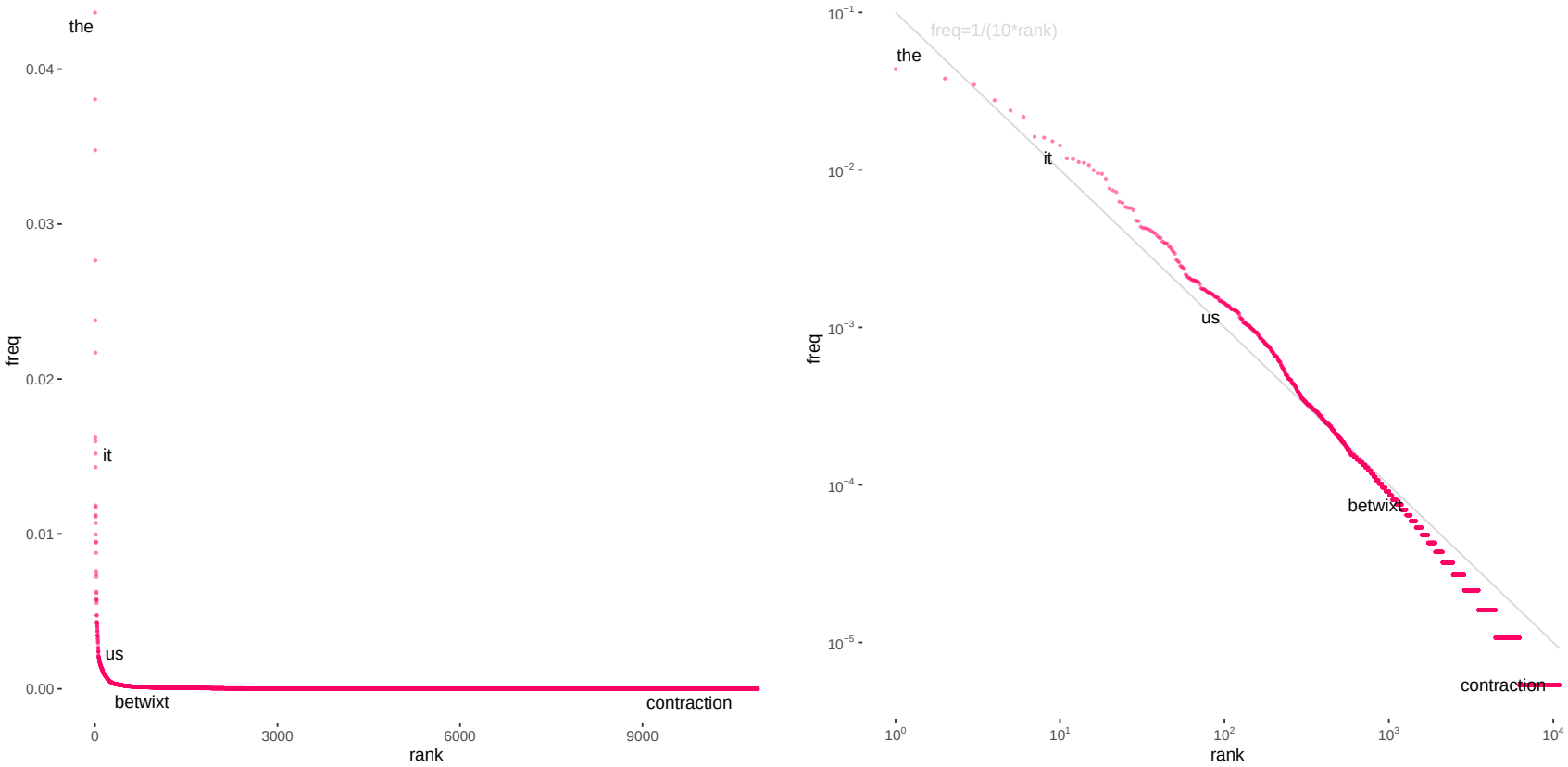
Sketch of how I checked

First we need to turn absolute frequencies into relative frequencies. This isn't too hard. We know our total word count, so we just need to use `map` to divide each absolute frequency by this word count:

```
relFreqs :: Int -> [(Int, String)] -> [(Double, String)]
relFreqs n xs = let f = (\(m, w) -> (ezDiv m n, w)) in
  map f xs -- n is the total word count
```

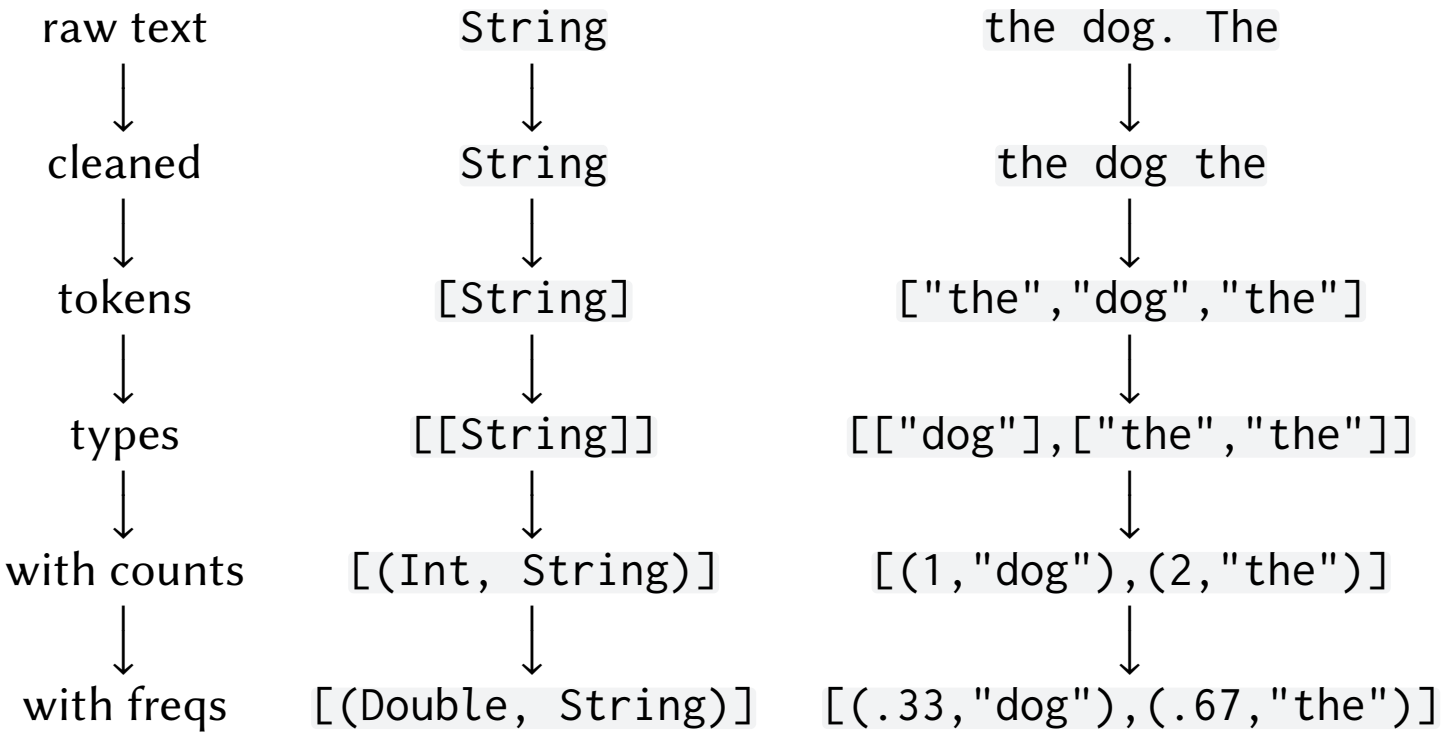
```
*Main> finally = relFreqs (length tokens) sortedByCount
*Main> last finally
(4.3651623068636 75e-2, "the")
-- ~4.4%
```

Zipfian distribution for *Great Expectations*



Some word types are (much much much) more common than others.
Zipf's Law: the n^{th} word is $\approx k$ times as frequent as the $k \times n^{th}$.

Summary


raw text
cleaned
tokens
types
with counts
with freqs
String
String
[String]
[[String]]
[(Int, String)]
[(Double, String)]
the dog. The
the dog the
["the","dog","the"]
[["dog"],["the","the"]]
[(1,"dog"),(2,"the")]
[(.33,"dog"),(.67,"the")]

In code

```haskell
main :: IO ()
main = do
  raw <- readFile "aux/great-expectations.txt"
  let cleaned = clean raw
  let tokens = tokenize cleaned
  let types = getTypes tokens
  let withCounts = sort (map addCount types)
  let finally = relFreqs (length tokens) withCounts
  writeFile "aux/freqs.txt" (unlines (map show finally))
  -- don't worry about how this last line works
```